Proceedings of the

Workshop on Algorithmic Aspects of Advanced Programming Languages

WAAAPL'99

Paris, France

September 30, 1999

Chris Okasaki, *Editor*

Department of Computer Science
Columbia University

Technical Report CUCS–023–99

Foreword

The first Workshop on Algorithmic Aspects of Advanced Programming Languages was held on September 30, 1999, in Paris, France, in conjunction with the PLI'99 conferences and workshops.

The choice of programming language has a huge effect on the algorithms and data structures that are to be implemented in that language. Traditionally, algorithms and data structures have been studied in the context of imperative languages. This workshop considers the algorithmic implications of choosing an advanced functional or logic programming language instead.

A total of eight papers were selected for presentation at the workshop, together with an invited lecture by Robert Harper.

We would like to thank Dider Rémy, general chair of PLI'99, for his assistance in organizing this workshop.

1999 Workshop on Algorithmic Aspects of Advanced Programming Languages

Paris, France
September 30, 1999

Invited Talk: 9:45–10:45

Session 1: 11:15–12:45

Session 2: 14:30–16:00

Session 3: 16:30–17:30

Manufacturing Datatypes

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract

This paper describes a general framework for designing purely functional datatypes that automatically satisfy given size or structural constraints. Using the framework we develop implementations of different matrix types (eg square matrices) and implementations of several tree types (eg Braun trees, 2-3 trees). Consider, for instance, representing square $n \times n$ matrices. The usual representation using lists of lists fails to meet the structural constraints: there is no way to ensure that the outer list and the inner lists have the same length. The main idea of our approach is to solve in a first step a related, but simpler problem, namely to generate the multiset of all square numbers. In order to describe this multiset we employ recursion equations involving finite multisets, multiset union, addition and multiplication lifted to multisets. In a second step we mechanically derive datatype definitions from these recursion equations, which enforce the 'squareness' constraint. The transformation makes essential use of polymorphic types.

1 Introduction

Many information structures are defined by certain size or structural constraints. Take, for instance, the class of perfectly balanced, binary leaf trees [10] (perfect leaf trees for short): a perfect leaf tree of height 0 is a leaf and a perfect leaf tree of height $h + 1$ is a node with two children, each of which is a perfect leaf tree of height h. How can we represent perfect leaf trees of arbitrary height such that the structural constraints are enforced? The usual recursive representation of binary leaf trees is apparently not very helpful since there is no way to ensure that the children of a node have the same height. As another example, consider square $n \times n$ matrices [14]. How do we represent square matrices such that the matrices are actually square? Again, the standard representation using lists of lists fails to meet the constraints: the outer list and the inner lists have not necessarily the same length. In this paper, we present a framework that allows to design representations of perfect leaf trees, square matrices, and many other information structures that automatically satisfy the given size or structural constraints.

Let us illustrate the main ideas by means of example. As a first example, we will devise a representation of *Toeplitz matrices* [6] where a Toeplitz matrix is an $n \times n$ matrix (a_{ij}) such that $a_{ij} = a_{i-1,j-1}$ for $1 < i, j \leqslant n$. Clearly, to represent a Toeplitz matrix of size $n + 1$ it suffices to store $2 * n + 1$ elements. Now, instead of designing a representation from scratch we first solve a related, but apparently simpler problem, namely, to generate the set of all odd numbers. Actually, we will work with multisets instead of sets for reasons to be explained later. In order to describe multisets of natural numbers we employ systems of recursion equations. The following system, for instance, specifies the multiset of all odd numbers, ie the multiset which contains

one occurrence of each odd number.

$$odd \quad = \quad \{\!|1|\!\} \uplus \{\!|2|\!\} + odd$$

Here, $\{\!|n|\!\}$ denotes the singleton multiset that contains n exactly once, $(\uplus)$ denotes multiset union and $(+)$ is addition lifted to multisets: $A + B = \{\!|a + b \mid a \leftarrow A; b \leftarrow B|\!\}$. We agree upon that $(+)$ binds more tightly than $(\uplus)$. Now, how can we turn the equation into a sensible datatype definition for Toeplitz matrices? The first thing to note is that we are actually looking for a datatype that is parameterized by the type of matrix elements. Such a type is also known as a *type constructor* or as a *functor*[1]. An element of a parameterized type is called a *container*. The equation above has the following counterpart in the world of functors.

$$Odd \quad = \quad Id \mid (Id \times Id) \times Odd$$

Here, *Id* is the identity functor given by $Id\ a = a$. Furthermore, $(|)$ and $(\times)$ denote disjoint sums and products lifted to functors, ie $(F_1 \mid F_2)\ a = F_1\ a \mid F_2\ a$ and $(F_1 \times F_2)\ a = F_1\ a \times F_2\ a$. Comparing the two equations we see that $\{\!|1|\!\}$ corresponds to *Id*, $(\uplus)$ corresponds to $(|)$, and $(+)$ corresponds to $(\times)$. This immediately implies that $Id \times Id$ corresponds to $\{\!|1|\!\} + \{\!|1|\!\} = \{\!|2|\!\}$. The relationship is very tight: the functor corresponding to a multiset M contains, for each member of M, a container of that size. For instance, $Id \times Id$ corresponds to $\{\!|1|\!\} + \{\!|1|\!\} = \{\!|2|\!\}$ as it contains one container of size 2; $Id \mid Id \times Id$ corresponds to $\{\!|1|\!\} \uplus \{\!|1|\!\} + \{\!|1|\!\} = \{\!|1, 2|\!\}$ as it contains one container of size 1 and another one of size 2.

Functor equations are written in a compositional style. To derive a datatype declaration from a functor equation we simply rewrite it into an applicative form—additionally adding constructor names and possibly making cosmetic changes.[2]

$$\textbf{data}\ Toeplitz\ a \quad = \quad Corner\ a \mid Extend\ a\ a\ (Toeplitz\ a)$$

The left upper corner of a Toeplitz matrix is represented by *Corner a*; *Extend r c m* extends the matrix *m* by an additional row and an additional column, both of which are represented by elements. For instance, the 4×4 Toeplitz matrix (a_{ij}) is represented by

$$Extend\ a_{41}\ a_{14}\ (Extend\ a_{31}\ a_{13}\ (Extend\ a_{21}\ a_{12}\ (Corner\ a_{11}))) \ .$$

Of course, this is not the only implementation conceivable. Alternatively, we can define *odd* in terms of the set of all even numbers.

$$\begin{aligned} odd &= \{\!|1|\!\} + even \\ even &= \{\!|0|\!\} \uplus \{\!|2|\!\} + even \end{aligned}$$

As innocent as this variation may look it has the advantage that the left upper corner can be accessed in constant time as opposed to linear time with the first representation.

$$\begin{aligned} \textbf{data}\ Toeplitz\ a &= Toeplitz\ a\ (List2\ a) \\ \textbf{data}\ List2\ a &= Nil2 \mid Cons2\ a\ a\ (List2\ a) \end{aligned}$$

Easier still, we may define *odd* in terms of the natural numbers using the fact that each odd number is of the form $1 + n * 2$ for some n.

$$\begin{aligned} odd &= \{\!|1|\!\} + nat * \{\!|2|\!\} \\ nat &= \{\!|0|\!\} \uplus \{\!|1|\!\} + nat \end{aligned}$$

[1] Categorically speaking, a functor must satisfy additional conditions, see [3]. All the type constructors listed in this paper are functors in the category-theoretical sense.

[2] Examples are given in the functional language Haskell 98 [15].

The first equation makes use of the multiplication operation, which is defined analogously to $(+)$. To which operation on functors does multiplication correspond? We will see that under certain conditions to be spelled out later $(*)$ corresponds to the composition of functors $(\cdot)$ given by $(F_1 \cdot F_2)\ a = F_1\ (F_2\ a)$. The functor equations derived from *odd* and *nat* are

$$\begin{aligned} Odd &= Id \times Nat \cdot (Id \times Id) \\ Nat &= K\ Unit \mid Id \times Nat\ . \end{aligned}$$

Here, $K\ t$ denotes the constant functor given by $K\ t\ a = t$ and *Unit* is the unit type containing a single element. Note that $K\ Unit$ corresponds to $\wr 0 \wr$. Unsurprisingly, *Nat* models the ubiquitous datatype of polymorphic lists.

$$\begin{aligned} \textbf{data}\ Toeplitz\ a &= Toeplitz\ a\ (List\ (a, a)) \\ \textbf{data}\ List\ a &= Nil \mid Cons\ a\ (List\ a) \end{aligned}$$

Thus, to store an even number of elements we simply use a list of pairs. This representation has the advantage that the list type can be easily replaced by a more efficient sequence type.

Next, let us apply the technique to design a representation of perfect leaf trees. The related problem is simple: we have to generate the multiset of all powers of 2.

$$power = \wr 1 \wr \uplus power * \wr 2 \wr$$

The corresponding functor equation is

$$Power = Id \mid Power \cdot (Id \times Id)\ ,$$

from which we can easily derive the following datatype definition.

$$\textbf{data}\ Perfect\ a = Zero\ a \mid Succ\ (Perfect\ (a, a))$$

Thus, a perfect leaf tree of height 0 is a leaf and a perfect leaf tree of height $h + 1$ is a perfect leaf tree of height h, whose leaves contain pairs of elements. Note that this definition proceeds *bottom-up* whereas the definition given in the beginning proceeds *top-down.* The type *Perfect* is an example for a so-called *nested datatype* [4]: the recursive call of *Perfect* on the right-hand side is not a copy of the declared type on the left-hand side, ie the type recursion is nested.

As the final example, let us tackle the problem of representing square matrices. We soon find that the related problem of generating the multiset of all square numbers is not quite as easy as before. One could be tempted to define $square = nat * nat$. However, this does not work since the resulting multiset contains products of arbitrary numbers. Incidentally, $nat * nat$ is related to $List \cdot List$, the lists of lists implementation we already rejected. We must somehow arrange that $(*)$ is only applied to singleton multisets. A trick to achieve this is to first rewrite the definition of *nat* into a *tail-recursive* form.

$$\begin{aligned} nat &= nat'\ \wr 0 \wr \\ nat'\ n &= n \uplus nat'\ (\wr 1 \wr + n) \end{aligned}$$

The definition of nat' closely resembles the function $from :: Int \to [Int]$ given by $from\ n = n : from\ (n + 1)$, which generates the infinite list of successive integers beginning with n. Now, to obtain square numbers we simply replace n by $n * n$ in the second equation.

$$\begin{aligned} square &= square'\ \wr 0 \wr \\ square'\ n &= n * n \uplus square'\ (\wr 1 \wr + n) \end{aligned}$$

Using this trick we are, in fact, able to enumerate the codomain of an arbitrary polynomial. Even more interesting, this trick is applicable to other representations of sequences, as well. But, we are skipping ahead. For now, let us determine the datatypes corresponding to *square* and *square′*. From the functor equations

$$\begin{array}{lcl} \mathit{Square} & = & \mathit{Square}'\ (K\ \mathit{Unit}) \\ \mathit{Square}'\ f & = & f \cdot f \mid \mathit{Square}'\ (\mathit{Id} \times f) \end{array}$$

we can derive the following datatype declarations.

$$\begin{array}{lcl} \textbf{type}\ \mathit{Matrix}\ a & = & \mathit{Matrix}'\ \mathit{Nil}\ a \\ \textbf{data}\ \mathit{Matrix}'\ t\ a & = & \mathit{Zero}\ (t\ (t\ a)) \mid \mathit{Succ}\ (\mathit{Matrix}'\ (\mathit{Cons}\ t)\ a) \\ \textbf{data}\ \mathit{Nil}\ a & = & \mathit{Nil} \\ \textbf{data}\ \mathit{Cons}\ t\ a & = & \mathit{Cons}\ a\ (t\ a) \end{array}$$

The type constructors *Nil* and *Cons t* correspond to $K\ \mathit{Unit}$ and $\mathit{Id} \times f$. As an aside, note that *Nil* and *Cons* are obtained by decomposing the *List* datatype into a base and into a recursive case. Furthermore, note that *Square′* is not a functor but a *higher-order functor* as it takes functors to functors. Accordingly, *Matrix′* is a type constructor of kind $(* \to *) \to (* \to *)$. Recall that the kind system of Haskell specifies the 'type' of a type constructor [12]. The '$*$' kind represents nullary constructors like *Bool* or *Int*. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. Though the type of square matrices looks daunting, it is comparatively easy to construct elements of that type. Here is a square matrix of size 3.

$$\begin{array}{l} \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\mathit{Cons}\ (\mathit{Cons}\ a_{11}\ (\mathit{Cons}\ a_{12}\ (\mathit{Cons}\ a_{13}\ \mathit{Nil})))) \\ \qquad\qquad\qquad\qquad (\mathit{Cons}\ (\mathit{Cons}\ a_{21}\ (\mathit{Cons}\ a_{22}\ (\mathit{Cons}\ a_{23}\ \mathit{Nil})))) \\ \qquad\qquad\qquad\qquad (\mathit{Cons}\ (\mathit{Cons}\ a_{31}\ (\mathit{Cons}\ a_{32}\ (\mathit{Cons}\ a_{33}\ \mathit{Nil})))) \\ \qquad\qquad\qquad\qquad (\mathit{Nil}))))))) \end{array}$$

Perhaps surprisingly, we have essentially a list of lists! The only difference to the standard representation is that the size of the matrix is additionally encoded into a prefix of *Zero* and *Succ* constructors. It is this prefix that takes care of the size constraints.

This completes the overview. The rest of the paper is organized as follows. Section 2 introduces multisets and operations on multisets. Furthermore, we show how to transform equations into a tail-recursive form. Section 3 explains functors and makes the relationship between multisets and functors precise. A multitude of examples is presented in Section 4: among other things we study random-access lists, Braun trees, 2-3 trees, and square matrices. Finally, Section 5 reviews related work and points out directions for future work.

2 Multisets

A multiset of type $\{\!|a|\!\}$ is a collection of elements of type a that takes account of their number but not of their order. In this paper, we will only consider multisets formed according to the following grammar.

$$M \quad ::= \quad \varnothing \mid \{\!|0|\!\} \mid \{\!|1|\!\} \mid (M \uplus M) \mid (M + M) \mid (M * M)$$

Here, $\varnothing$ denotes the empty multiset, $\{\!|n|\!\}$ denotes the singleton multiset that contains n exactly once, $(\uplus)$ denotes multiset union, $(+)$ and $(*)$ are addition and multiplication lifted to multisets, ie they are defined by $A \otimes B = \{\!|a \otimes b \mid a \leftarrow A; b \leftarrow B|\!\}$ for $\otimes \in \{+, *\}$. If the meaning can be resolved from the context, we abbreviate $\{\!|n|\!\}$ by n. Furthermore, we agree upon that multiplication takes precedence over addition, which in turn takes precedence over multiset union.

$$
\begin{array}{rcl}
\{\!|m|\!\} + \{\!|n|\!\} & = & \{\!|m+n|\!\} \\
\{\!|m|\!\} * \{\!|n|\!\} & = & \{\!|m*n|\!\}
\end{array}
\qquad
\begin{array}{rcl}
A \uplus (B \uplus C) & = & (A \uplus B) \uplus C \\
A \uplus B & = & B \uplus A
\end{array}
$$

$$
\begin{array}{rcl}
A + (B + C) & = & (A + B) + C \\
A + B & = & B + A \\
0 + A & = & A
\end{array}
\qquad
\begin{array}{rcl}
\varnothing \uplus A & = & A \\
\varnothing + A & = & \varnothing \\
\varnothing * A & = & \varnothing
\end{array}
$$

$$
\begin{array}{rcl}
A * (B * C) & = & (A * B) * C \\
a * b & = & b * a \\
1 * A & = & A \\
A * 1 & = & A
\end{array}
\qquad
\begin{array}{rcl}
(A \uplus B) + C & = & A + C \uplus B + C \\
(A \uplus B) * C & = & A * C \uplus B * C \\
(A + B) * c & = & A * c + B * c \\
0 * A & = & 0
\end{array}
$$

A, B, C are multisets $\quad$ a, b, c are simple multisets $\quad$ m, n are natural numbers

Figure 1: Laws of the operations.

Multisets are defined by *higher-order recursion equations*. Higher-order means that the equations may not only involve multisets, but also functions over multisets, function over functions over multisets etc. In this paper, we will, however, restrict ourselves to first-order equations. The exploration of higher-order kinds is the topic of future research. The meaning of higher-order recursion equations is given by the usual least fixpoints semantics.

A multiset is called *simple* iff it is either the empty multiset or a multiset containing a single element arbitrarily often. Simple multiset are denoted by lower case letters. A product $A * B$ is called *admissible* iff B denotes a simple multiset. For instance, $nat * 2$ is admissible while $nat * nat$ is not. We will see in Section 3 that only admissible products correspond to compositions of functors. That is, $nat * 2$ corresponds to $Nat \cdot (Id \times Id)$ but $nat * nat$ does not correspond to $Nat \cdot Nat$. For that reason, we confine ourselves to admissible products when defining multisets.

A multiset is called *unique* iff each element occurs at most once. For instance, the multiset *pos* given by $pos = 1 \uplus 1 + pos$ is unique whereas $pos = 1 \uplus pos + pos$ denotes a non-unique multiset. Note that the first definition corresponds to non-empty lists and the second to leaf trees. The ability to distinguish between unique and non-unique representations is the main reason for using multisets instead of sets.

The multiset operations satisfy a variety of laws listed in Figure 1. The laws have been chosen so that they hold both for multisets *and* for the corresponding operations on functors. This explains why, for instance, $a * b = b * a$ is restricted to simple multisets: the corresponding property on functors, $F \cdot G = G \cdot F$, does not hold in general. It is valid, however, if G only comprises containers of one size. Of course, for functors the equations state isomorphisms rather than equalities.

In the introduction we have transformed the recursive definition of the multiset of all natural numbers into a tail-recursive form. In the rest of this section we will study this transformation in more detail. A function $h :: \{\!|a|\!\} \to \{\!|a|\!\}$ on multisets is said to be a *homomorphism* iff $h\ \varnothing = \varnothing$ and $h\ (A \uplus B) = h\ A \uplus h\ B$. For instance, $h\ N = A + N * b$ is a homomorphism while $g\ N = N + N$ is not. Let $h_1, \ldots, h_n$ be homomorphisms, let A be a multiset, and let X be given by

$$X \quad = \quad A \uplus h_1\ X \uplus \cdots \uplus h_n\ X \ .$$

The definition of X is not tail-recursive as the recursive occurrences of X are nested inside function calls. Note that *nat* is an instance of this scheme with $A = \{\!|0|\!\}$, $n = 1$, and $h_1\ N = \{\!|1|\!\} + N$. Now, the *tail-recursive*

variant of X is $f\ A$ with f given by

$$f\ N \quad = \quad N \uplus f\ (h_1\ N) \uplus \cdots \uplus f\ (h_n\ N)\ .$$

The definition of f is called *tail-recursive* for obvious reasons. Note that $nat'\ \{\!|0|\!\}$ is the tail-recursive variant of *nat*. The correctness of the transformation is implied by the following theorem.

Theorem 1 *Let $X :: \{\!|a|\!\}$, $A :: \{\!|a|\!\}$, and $f :: \{\!|a|\!\} \to \{\!|a|\!\}$ be given as above, then $X = f\ A$.*

3 Functors

In close analogy to multiset expressions we define the syntax of *functor expressions* by the following grammar.

$$F \quad ::= \quad K\ Void \mid K\ Unit \mid Id \mid (F \mid F) \mid (F \times F) \mid (F \cdot F)$$

Here, $K\ t$ denotes the constant functor given by $K\ t\ a = t$, *Void* is the empty type, and *Unit* is the unit type containing a single element. By *Id* we denote the identity functor given by $Id\ a = a$; $F_1 \cdot F_2$ denotes functor composition given by $(F_1 \cdot F_2)\ a = F_1\ (F_2\ a)$. Disjoint sums and products are defined pointwise: $(F_1 \mid F_2)\ a = F_1\ a \mid F_2\ a$ and $(F_1 \times F_2)\ a = F_1\ a \times F_2\ a$.

All these constructs can be easily defined in Haskell. First of all, we require the following type definitions.

$$\begin{array}{lcl} \textbf{type}\ Unit & = & () \\ \textbf{data}\ Either\ a_1\ a_2 & = & Left\ a_1 \mid Right\ a_2 \\ \textbf{data}\ (a_1, a_2) & = & (a_1, a_2) \end{array}$$

The predefined types $Either\ a_1\ a_2$ and (a_1, a_2) implement disjoint sums and products. The operations on functors are then defined by

$$\begin{array}{lcl} \textbf{newtype}\ Id\ a & = & Id\ a \\ \textbf{newtype}\ K\ a\ b & = & K\ a \\ \textbf{newtype}\ Sum\ t_1\ t_2\ a & = & Sum\ (Either\ (t_1\ a)\ (t_2\ a)) \\ \textbf{newtype}\ Prod\ t_1\ t_2\ a & = & Prod\ (t_1\ a, t_2\ a) \\ \textbf{newtype}\ Comp\ t_1\ t_2\ a & = & Comp\ (t_1\ (t_2\ a))\ . \end{array}$$

Using these type constructors it is straightforward to translate a functor equation into a Haskell datatype definition. For reasons of readability, we will often define special instances of the general schemes writing *Nil* instead of *K Unit* or *Cons t* instead of *Prod Id t*.

The translation of multisets into functors is given by the following table.

m_1	m_2	$\varnothing$	$\{\!\|0\|\!\}$	$\{\!\|1\|\!\}$	$m_1 \uplus m_2$	$m_1 + m_2$	$m_1 * m_2$
f_1	f_2	$K\ Void$	$K\ Unit$	Id	$f_1 \mid f_2$	$f_1 \times f_2$	$f_1 \cdot f_2$

We say that F *corresponds to* M if F is obtained from M using this translation. In the rest of this section we will briefly sketch the correctness of the translation. Informally, the functor corresponding to a multiset M contains, for each member of M, a container of that size. This statement can be made precise using the framework of polytypic programming [11]. Briefly, a polytypic function is one that is defined by induction on the structure of functor expressions. A simple example for a polytypic function is $sum\langle f\rangle :: f\ \mathbb{N} \to \mathbb{N}$, which sums a structure of natural numbers. To make the relationship between multisets and functors precise we furthermore require the function $fan\langle f\rangle :: a \to \{\!|f\ a|\!\}$, which generates the multiset of all structures of type $f\ a$ from a given seed of type a. For instance, $fan\langle List\rangle\ 1$ generates the multiset of all lists that contain 1 as the single element.

Theorem 2 *If the functor F corresponds to the multiset M and if M's definition only involves admissible products, then $M = \wr sum\langle F\rangle\ a \mid a \leftarrow fan\langle F\rangle\ 1\wr$.*

The following example shows that it is necessary to restrict products to admissible products: if we compose the functors corresponding to $\wr 1, 2\wr$ and $\wr 1, 3\wr$, we obtain a functor that corresponds to $\wr 1, 2, 3, 4, 4, 6\wr$. In general, functor composition corresponds to the multiset operation $(\circledast)$ given by

$$A \circledast B \quad = \quad \wr b_1 + \cdots + b_a \mid a \leftarrow A; b_1 \leftarrow B; \ldots; b_a \leftarrow B\wr \ .$$

We take a container of type A and fill each of its slots with a container of type B. Summing the sizes of the B containers yields the overall size. The operations $(*)$ and $(\circledast)$ coincide only for admissible products, ie if the containers of type B all have equal size.

4 Examples

In this section we apply the framework to generate efficient implementations of vectors (aka lists or sequences or arrays) and matrices.

4.1 Lists

A vector or a sequence type contains containers of arbitrary size. The problem related to designing a sequence type is, of course, to generate the multiset of all natural numbers. Different ways to describe this set correspond to different implementations of vectors. Perhaps surprisingly, there is an abundance of ways to solve this problem. In the introduction we already encountered the most direct solution:

$$nat_0 \quad = \quad 0 \uplus 1 + nat_0 \ .$$

If we transform the corresponding functor equation

$$Nat_0 \quad = \quad K\ Unit \mid Id \times Nat_0$$

into a Haskell datatype, we obtain the ubiquitous datatype of polymorphic lists.

$$\textbf{data}\ Vector\ a \quad = \quad Nil \mid Cons\ a\ (Vector\ a)$$

As an example, the list representation of the vector $(0, 1, 2, 3, 4, 5)$ is

$$Cons\ 0\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ (Cons\ 4\ (Cons\ 5\ Nil))))) \ .$$

The tail-recursive variant of nat_0 is given by

$$\begin{array}{lcl} nat_1 & = & nat_1'\ 0 \\ nat_1'\ n & = & n \uplus nat_1'\ (1+n) \ . \end{array}$$

From the functor equations

$$\begin{array}{lcl} Nat_1 & = & Nat_1'\ (K\ Unit) \\ Nat_1'\ f & = & f \mid Nat_1'\ (Id \times f) \end{array}$$

we can derive the following datatype definitions.

$$\begin{array}{lcl} \textbf{type}\ Vector & = & Vector'\ Nil \\ \textbf{data}\ Vector'\ t\ a & = & Zero\ (t\ a) \mid Succ\ (Vector'\ (Cons\ t)\ a) \end{array}$$

Using this representation the vector $(0,1,2,3,4,5)$ is written somewhat lengthy as

$$\begin{aligned}&\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\\ &\quad \mathit{Cons}\ 0\ (\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ (\mathit{Cons}\ 5\ \mathit{Nil}))))))))))))\ .\end{aligned}$$

Fortunately, we can simplify the definitions slightly. Recall that *Vector′* is a type of kind $(*\to *)\to(*\to *)$. In this case the 'higher-orderness' is, however, not required. Noting that the first argument of *Vector′* is always applied to the second we can transform *Vector′* into a first-order functor of kind $*\to *\to *$.

$$\begin{aligned}\textbf{type}\ \mathit{Vector} &= \mathit{Vector'}\ ()\\ \textbf{data}\ \mathit{Vector'}\ t\ a &= \mathit{Zero}\ t \mid \mathit{Succ}\ (\mathit{Vector'}\ (a,t)\ a)\end{aligned}$$

The two variants of *Vector′* are related by $\mathit{Vector'_{ho}}\ t\ a = \mathit{Vector'_{fo}}\ (t\ a)\ a$ and $\mathit{Vector'_{fo}}\ t\ a = \mathit{Vector'_{ho}}\ (K\ t)\ a$. Note that the type *Matrix′* defined in the introduction is not amenable to this transformation since the first argument of *Matrix′* is used at different instances. Using the first-order definition $(0,1,2,3,4,5)$ is represented by

$$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (0,(1,(2,(3,(4,(5,())))))))))))\ .$$

4.2 Random-access lists

The definition of nat_0 is based on the unary representation of the natural numbers: a natural number is either zero or the successor of a natural number. Of course, we can also base the definition on the binary number system: a natural number is either zero, even, or odd.

$$\mathit{nat}_2 \quad = \quad 0 \uplus \mathit{nat}_2 * 2 \uplus 1 + \mathit{nat}_2 * 2$$

Transforming the corresponding functor equation

$$\mathit{Nat}_2 \quad = \quad K\ \mathit{Unit} \mid \mathit{Nat}_2 \cdot (\mathit{Id} \times \mathit{Id}) \mid \mathit{Id} \times \mathit{Nat}_2 \cdot (\mathit{Id} \times \mathit{Id})$$

into a Haskell datatype yields

$$\textbf{data}\ \mathit{Vector}\ a \quad = \quad \mathit{Null} \mid \mathit{Zero}\ (\mathit{Vector}\ (a,a)) \mid \mathit{One}\ a\ (\mathit{Vector}\ (a,a))\ .$$

Interestingly, this definition implements *random-access lists* [13], which support logarithmic access to individual vector elements. A random-access list is basically a sequence of perfect leaf trees of increasing height. The vector $(0,1,2,3,4,5)$, for instance, is represented by

$$\mathit{Zero}\ (\mathit{One}\ (0,1)\ (\mathit{One}\ ((2,3),(4,5))\ \mathit{Null}))\ .$$

The sequence of *Zero* and *One* constructors encodes the size of the vector in binary representation (with the least significant bit first): we have $(011)_2 = 6$. The representation of a vector of size 11 is depicted in Figure 2(a). Note that the representation is not unique because of leading zeros: the empty sequence, for example, can be represented by *Null*, *Zero Null*, *Zero* (*Zero Null*) etc. There are at least two ways to repair this defect. The following definition ensures that the leading digit is always a one.

$$\begin{aligned}\mathit{nat}_3 &= 0 \uplus \mathit{pos}_3\\ \mathit{pos}_3 &= 1 \uplus \mathit{pos}_3 * 2 \uplus 1 + \mathit{pos}_3 * 2\end{aligned}$$

More elegantly, one can define a *zeroless representation* [13], which employs the digits 1 and 2 instead of 0 and 1. We call this variant of the binary number system *1-2 system*.

$$\mathit{nat}_4 \quad = \quad 0 \uplus 1 + \mathit{nat}_4 * 2 \uplus 2 + \mathit{nat}_4 * 2$$

This alternative has the further advantage that accessing the i-th element runs in $O(\log i)$ time [13].

4.3 Fork-node trees

Now, let us transform nat_3 into a tail-recursive form.

$$\begin{array}{lcl} nat_5 & = & 0 \uplus pos'_5\ 1 \\ pos'_5\ n & = & n \uplus pos'_5\ (n*2) \uplus pos'_5\ (1+n*2) \end{array}$$

Note that we may replace $n*2$ by $2*n = n+n$ if pos'_5 is called with a simple multiset as in $pos'_5\ 1$. The corresponding functor equations look puzzling.

$$\begin{array}{lcl} Nat_5 & = & K\ Unit \mid Pos'_5\ Id \\ Pos'_5\ f & = & f \mid Pos'_5\ (f \cdot (Id \times Id)) \mid Pos'_5\ (Id \times f \cdot (Id \times Id)) \end{array}$$

In order to improve the readability of the derived datatypes let us define idioms for $2*n = n+n$ and $1+2*n = 1+n+n$.

$$\begin{array}{lcl} \textbf{data}\ Fork\ t\ a & = & Fork\ (t\ a)\ (t\ a) \\ \textbf{data}\ Node\ t\ a & = & Node\ a\ (t\ a)\ (t\ a) \end{array}$$

These definitions assume that t is a simple functor. The alternative definitions **newtype** $Fork\ t\ a = Fork\ (t\ (a,a))$ and **data** $Node\ t\ a = Node\ a\ (t\ (a,a))$, which correspond to $n*2$ and $1+n*2$, work for arbitrary functors but are more awkward to use. Building upon *Fork* and *Node* the Haskell datatypes read

$$\begin{array}{lcl} \textbf{data}\ Vector\ a & = & Empty \mid NonEmpty\ (Vector'\ Id\ a) \\ \textbf{data}\ Vector'\ t\ a & = & Base\ (t\ a) \\ & \mid & Zero\ (Vector'\ (Fork\ t)\ a) \\ & \mid & One\ (Vector'\ (Node\ t)\ a)\ . \end{array}$$

A vector of size n is represented by a complete binary tree of height $\lfloor \log_2 n \rfloor + 1$. A node in the i-th level of this tree is labelled with an element iff the i-th digit in the binary decomposition of n is one. The lowest level, which corresponds to a leading one, always contains elements. To the best of the author's knowledge this data structure, which we baptize *fork-node trees* for want of a better name, has not been described elsewhere.[3] Our running example, the vector $(0,1,2,3,4,5)$, is represented by

$$NonEmpty\ (One\ (Zero\ (Base\ (Fork\ (Node\ 0\ (Id\ 1)\ (Id\ 2))\ (Node\ 3\ (Id\ 4)\ (Id\ 5))))))\ .$$

Again, the size of the vector is encoded into the prefix of constructors: replacing *NonEmpty* and *One* by 1 and *Zero* by 0 yields the binary decomposition of the size *with the most significant bit first*. Figure 2(b) shows a sample vector of 11 elements. The vector elements are stored in left-to-right preorder: if the tree has a root, it contains the first element; the elements in the left tree precede the elements in the right tree. This layout is, however, by no means compelling. Alternatively, one can interleave the elements of the left and the right subtree: if l represents the vector $(b_1,\ldots,b_n)$ and r represents $(c_1,\ldots,c_n)$, then $Fork\ l\ r$ represents the vector $(b_1,c_1,\ldots,b_n,c_n)$ and $Node\ a\ l\ r$ represents $(a,b_1,c_1,\ldots,b_n,c_n)$. This choice facilitates the extension of a vector at the front and also slightly simplifies accessing a vector element.

As always for vector types we can 'firstify' the type definitions.

$$\begin{array}{lcl} \textbf{data}\ Vector\ a & = & Empty \mid NonEmpty\ (Vector'\ a\ a) \\ \textbf{data}\ Vector'\ t\ a & = & Base\ t \\ & \mid & Zero\ (Vector'\ (t,t)\ a) \\ & \mid & One\ (Vector'\ (a,t,t)\ a) \end{array}$$

[3] Since this paper was written, I have learned that Hongwei Xi has independently discovered the same data structure.

The representation of $(0,1,2,3,4,5)$ now consists of nested pairs and triples.

$$\mathit{NonEmpty}\ (\mathit{One}\ (\mathit{Zero}\ (\mathit{Base}\ ((0,1,2),(3,4,5)))))$$

Finally, let us remark that the tail-recursive variant of nat_4, which is based on the 1-2 system, yields a similar tree shape: a node on the i-th level contains d elements where d is the i-th digit in the 1-2 decomposition of the vector's size.

4.4 Rightist right-perfect trees

The definition of nat_2 is based on the fact that all natural numbers can be generated by shifting $(n*2)$ and setting the least significant bit $(1+n*2)$. The following definition sets bits at arbitrary positions by repeatedly shifting a one.

$$\begin{array}{lcl} nat_6 & = & nat_6'\ 1 \\ nat_6'\ p & = & 0 \uplus nat_6'\ (p*2) \uplus p + nat_6'\ (p*2) \end{array}$$

Of course, the two definitions are not unrelated, we have

$$nat_2 * p \quad = \quad nat_6'\ p\ ,$$

ie $nat_6'\ p$ generates all multiples of p. In the i-th level of recursion the parameter of nat_6' equals $p*2^i$ if the initial call was $nat_6'\ p$. Now, transforming the corresponding functor equations, which assume that f is simple,

$$\begin{array}{lcl} Nat_6 & = & Nat_6'\ Id \\ Nat_6'\ f & = & f \mid Nat_6'\ (f \times f) \mid f \times Nat_6'\ (f \times f) \end{array}$$

into Haskell datatypes yields

$$\begin{array}{lcl} \textbf{type}\ \mathit{Vector} & = & \mathit{Vector'}\ \mathit{Id} \\ \textbf{data}\ \mathit{Vector'}\ t\ a & = & \mathit{Null} \\ & \mid & \mathit{Zero}\ (\mathit{Vector'}\ (\mathit{Fork}\ t)\ a) \\ & \mid & \mathit{One}\ (t\ a)\ (\mathit{Vector'}\ (\mathit{Fork}\ t)\ a)\ . \end{array}$$

This datatype implements *higher-order random-access lists* [9]. If we 'firstify' the type constructor *Vector'*, we obtain the first-order variant as defined in Section 4.2. For a discussion of the tradeoffs we refer the interested reader to [9]. The vector $(0,1,2,3,4,5)$ is represented by

$$\mathit{Zero}\ (\mathit{One}\ (\mathit{Fork}\ (\mathit{Id}\ 0)\ (\mathit{Id}\ 1))\ (\mathit{One}\ (\mathit{Fork}\ (\mathit{Fork}\ (\mathit{Id}\ 2)\ (\mathit{Id}\ 3))\ (\mathit{Fork}\ (\mathit{Id}\ 4)\ (\mathit{Id}\ 6)))\ \mathit{Null}))\ .$$

Interestingly, using a slight generalization of Theorem 1 we can transform nat_6' into a tail-recursive form, as well.

$$\begin{array}{lcl} nat_7 & = & nat_7'\ 0\ 1 \\ nat_7'\ n\ p & = & n \uplus nat_7'\ n\ (p*2) \uplus nat_7'\ (n+p)\ (p*2) \end{array}$$

The function nat_7' is related to nat_2 by

$$n + nat_2 * p \quad = \quad nat_7'\ n\ p\ .$$

Assuming that p is simple we get the following functor equations

$$\begin{array}{lcl} Nat_7 & = & Nat_7'\ (K\ Unit)\ Id \\ Nat_7'\ f\ p & = & f \mid Nat_7'\ f\ (p \times p) \mid Nat_7'\ (f \times p)\ (p \times p)\ , \end{array}$$

from which we can easily derive the datatype definitions below.

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Vector} & = & \mathit{Vector'}\ (K\ \mathit{Unit})\ \mathit{Id} \\
\textbf{data}\ \mathit{Vector'}\ t\ p\ a & = & \mathit{Base}\ (t\ a) \\
 & | & \mathit{Even}\ (\mathit{Vector'}\ t\ (\mathit{Prod}\ p\ p)\ a) \\
 & | & \mathit{Odd}\ (\mathit{Vector'}\ (\mathit{Prod}\ t\ p)\ (\mathit{Prod}\ p\ p)\ a)
\end{array}$$

This datatype implements *rightist right-perfect trees* or *RR-trees* [7] where the offsprings of the nodes on the left spine form a sequence of perfect leaf trees of decreasing height. Note that if we change $\mathit{Prod}\ t\ p$ to $\mathit{Prod}\ p\ t$ in the last line, we obtain *leftist left-perfect trees.* Here is the vector $(0,1,2,3,4,5)$ written as an RR-tree.

$$\begin{array}{l}
\mathit{Even}\ (\mathit{Odd}\ (\mathit{Odd}\ (\mathit{Base}\ (\mathit{Prod}\ (\mathit{Prod}\ (K\ ()), \mathit{Prod}\ (\mathit{Id}\ 0, \mathit{Id}\ 1)), \\
\qquad\qquad \mathit{Prod}\ (\mathit{Prod}\ (\mathit{Id}\ 2, \mathit{Id}\ 3), \mathit{Prod}\ (\mathit{Id}\ 4, \mathit{Id}\ 5)))))))
\end{array}$$

Reading the constructors *Even* and *Odd* as digits (LSB first) gives the size of the vector. A sample vector of size 11 is shown in Figure 2(c). The 'firstification' of $\mathit{Vector'}$ is left as an exercise to the reader.

4.5 Braun trees

Let us apply the framework to design a representation of *Braun trees* [5]. Braun trees are node-oriented trees, which are characterized by the following balance condition: for all subtrees, the size of the left subtree is either exactly the size of the right subtree, or one element larger. In other words, a Braun tree of size $2*n+1$ has two children of size n and a Braun tree of size $2*n+2$ has a left child of size $n+1$ and a right child of size n. This motivates the following definition.

$$\begin{array}{lcl}
\mathit{braun} & = & \mathit{braun'}\ 0\ 1 \\
\mathit{braun'}\ n\ n' & = & n \uplus \mathit{braun'}\ (n+1+n)\ (n'+1+n) \\
 & & \quad \uplus \mathit{braun'}\ (n'+1+n)\ (n'+1+n')
\end{array}$$

The arguments of $\mathit{braun'}$ are always two successive natural numbers. From the corresponding functor equations

$$\begin{array}{lcl}
\mathit{Braun} & = & \mathit{Braun'}\ (K\ \mathit{Unit})\ \mathit{Id} \\
\mathit{Braun'}\ f\ f' & = & f \mid \mathit{Braun'}\ (f \times \mathit{Id} \times f)\ (f' \times \mathit{Id} \times f) \\
 & & \quad \mid \mathit{Braun'}\ (f' \times \mathit{Id} \times f)\ (f' \times \mathit{Id} \times f')
\end{array}$$

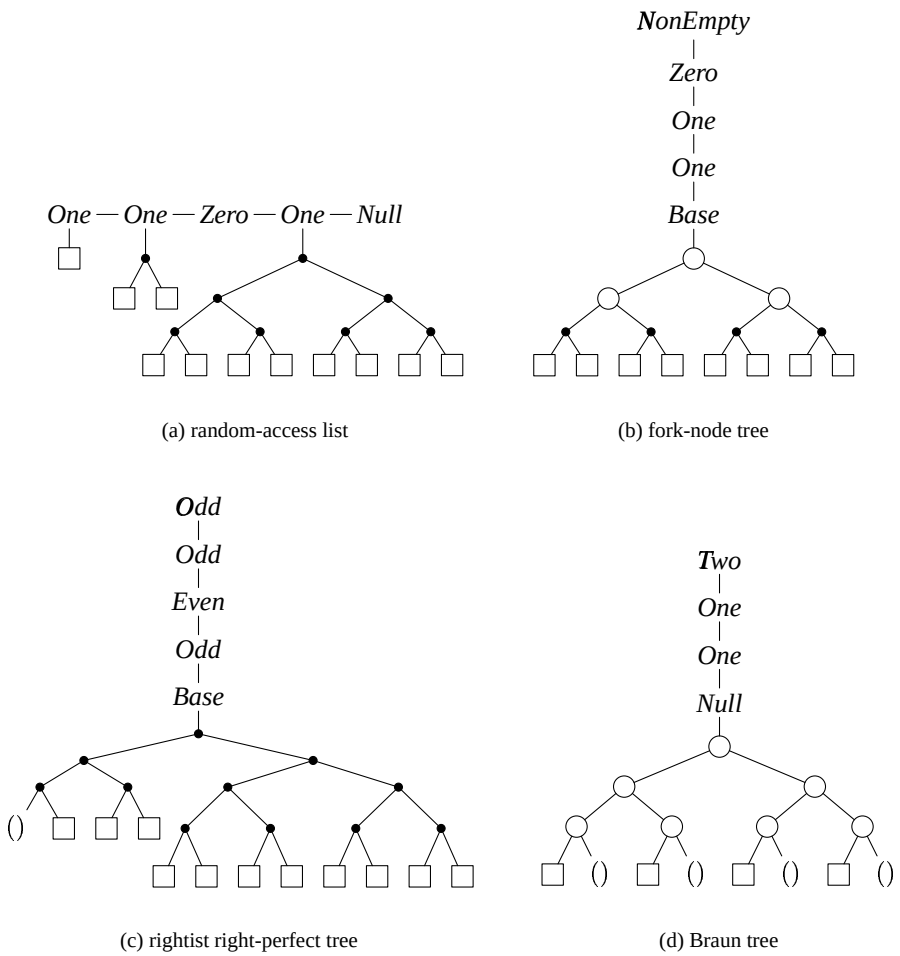
we can derive the following datatype definitions.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Bin}\ t_1\ t_2\ a & = & \mathit{Bin}\ (t_1\ a)\ a\ (t_2\ a) \\
\textbf{type}\ \mathit{Braun} & = & \mathit{Braun'}\ (K\ \mathit{Unit})\ \mathit{Id} \\
\textbf{data}\ \mathit{Braun'}\ t\ t'\ a & = & \mathit{Null}\ (t\ a) \\
 & | & \mathit{One}\ (\mathit{Braun'}\ (\mathit{Bin}\ t\ t)\ (\mathit{Bin}\ t'\ t)\ a) \\
 & | & \mathit{Two}\ (\mathit{Braun'}\ (\mathit{Bin}\ t'\ t)\ (\mathit{Bin}\ t'\ t')\ a)
\end{array}$$

Interestingly, Braun trees are based on the 1-2 number system (MSB first). The vector $(0,1,2,3,4,5)$, for instance, is represented as follows.

$$\mathit{Two}\ (\mathit{Two}\ (\mathit{Null}\ (\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Id}\ 0)\ 1\ (\mathit{Id}\ 2))\ 3\ (\mathit{Bin}\ (\mathit{Id}\ 4)\ 5\ (K\ ())))))$$

Figure 2(d) displays the representation of a vector of 11 elements. R. Paterson has described a similar implementation (personal communication).



(a) random-access list

(b) fork-node tree

(c) rightist right-perfect tree

(d) Braun tree

Figure 2: Different representations of a vector with 11 elements. Note that '□' represents a leaf (an element of *Id*), '•' an unlabelled node (an element of $Id \times Id$, *Fork t*, or *Prod* t_1 t_2), and '○' a labelled node (an element of *Node t* or *Bin* t_1 t_2).

4.6 2-3 trees

Up to now we have mainly considered unique representations where the shape of a data structure is completely determined by the number of elements it contains. Interestingly, unique representations are not well-suited for implementing search trees: one can prove a lower bound of $\Omega(\sqrt{n})$ for insertion and deletion in this case [16]. For that reason, popular search tree schemes such as 2-3 trees [2], red-black trees [8], or AVL-trees [1] are always based on non-unique representations. Let us consider how to implement, say, 2-3 trees. The other search tree schemes can be handled in an analogous fashion. The definition of 2-3 trees is similar to that of perfect leaf trees: a 2-3 tree of height 0 is a leaf and a 2-3 tree of height $h+1$ is a node with either two or three children, each of which is a 2-3 tree of height h. This similarity suggests to model 2-3 trees as follows.

$$\begin{array}{lcl} \mathit{tree23} & = & \mathit{tree23'}\ 0 \\ \mathit{tree23'}\ N & = & N \uplus \mathit{tree23'}\ (N+1+N \uplus N+1+N+1+N) \end{array}$$

Note that contrary to previous definitions the parameter of the auxiliary function does not range over simple sets. The corresponding functor equations

$$\begin{array}{lcl} \mathit{Tree23} & = & \mathit{Tree23'}\ (K\ \mathit{Unit}) \\ \mathit{Tree23'}\ F & = & F \mid \mathit{Tree23'}\ (F \times \mathit{Id} \times F \mid F \times \mathit{Id} \times F \times \mathit{Id} \times F) \end{array}$$

give rise to the following datatype definitions.

$$\begin{array}{lcl} \textbf{type}\ \mathit{Tree23}\ a & = & \mathit{Tree23'}\ \mathit{Nil}\ a \\ \textbf{data}\ \mathit{Tree23'}\ t\ a & = & \mathit{Zero}\ (t\ a) \mid \mathit{Succ}\ (\mathit{Tree23'}\ (\mathit{Node23}\ t)\ a) \\ \textbf{data}\ \mathit{Node23}\ t\ a & = & \mathit{Node2}\ (t\ a)\ a\ (t\ a) \mid \mathit{Node3}\ (t\ a)\ a\ (t\ a)\ a\ (t\ a) \end{array}$$

The vector $(0,1,2,3,4,5)$ has three different representations; one alternative is

$$\begin{array}{l} \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\mathit{Node3}\ (\mathit{Node3}\ \mathit{Nil}\ 0\ \mathit{Nil}\ 1\ \mathit{Nil})\ 2\ (\mathit{Node2}\ \mathit{Nil}\ 3\ \mathit{Nil}) \\ \qquad\qquad 4\ (\mathit{Node2}\ \mathit{Nil}\ 5\ \mathit{Nil})))) \ . \end{array}$$

Algorithms for insertion and deletion are described in [9].

4.7 Matrices

Let us finally design representations of square matrices and rectangular matrices. In the introduction we have already discussed the central idea: we take a tail-recursive definition of the natural numbers (or of the positive numbers)

$$\begin{array}{lcl} X & = & f\ a \\ f\ n & = & n \uplus f\ (h_1\ n) \uplus \cdots \uplus f\ (h_n\ n) \end{array}$$

and replace n by $n * n$ in the second equation:

$$\begin{array}{lcl} \mathit{square} & = & \mathit{square'}\ a \\ \mathit{square'}\ n & = & n * n \uplus \mathit{square'}\ (h_1\ n) \uplus \cdots \uplus \mathit{square'}\ (h_n\ n) \ . \end{array}$$

This transformation works provided a is a simple multiset and the h_i preserve simplicity. These conditions hold for all of the examples above with the notable exception of 2-3 trees. As a concrete example, here is an

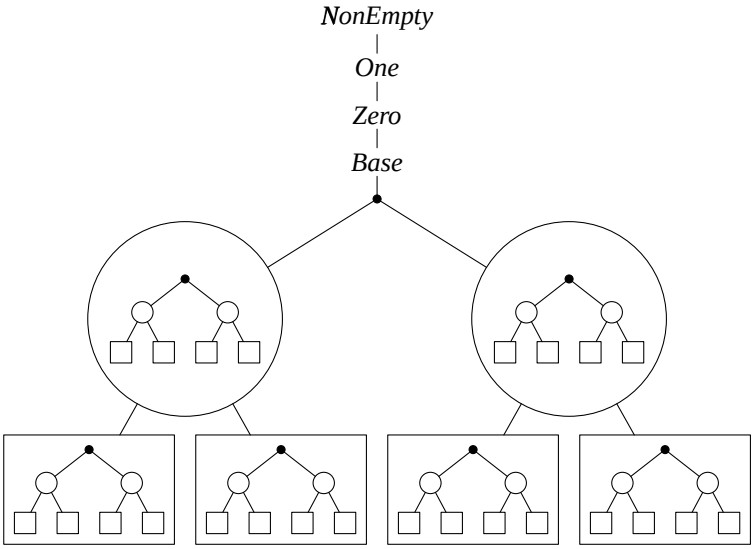


Figure 3: The representation of a 6×6 matrix based on fork-node trees.

implementation of square matrices based on fork-node trees.

$$\begin{array}{lcl} \textbf{data}\ \mathit{Matrix}\ a & = & \mathit{Empty} \mid \mathit{NonEmpty}\ (\mathit{Matrix}'\ \mathit{Id}\ a) \\ \textbf{data}\ \mathit{Matrix}'\ t\ a & = & \mathit{Base}\ (t\ (t\ a)) \\ & \mid & \mathit{Zero}\ (\mathit{Matrix}'\ (\mathit{Fork}\ t)\ a) \\ & \mid & \mathit{One}\ (\mathit{Matrix}'\ (\mathit{Node}\ t)\ a) \end{array}$$

The representation of a 6×6 matrix is shown in Figure 3.

Rectangular matrices are equally easy to implement. In this case we replace n by $\mathit{nat} * n$ in the second equation:

$$\begin{array}{lcl} \mathit{rect} & = & \mathit{rect}'\ a \\ \mathit{rect}'\ n & = & \mathit{nat} * n \uplus \mathit{rect}'\ (h_1\ n) \uplus \cdots \uplus \mathit{rect}'\ (h_n\ n)\ . \end{array}$$

Alternatively, one may use the following scheme.

$$\begin{array}{lcl} \mathit{rect} & = & \mathit{rect}'\ a\ a \\ \mathit{rect}'\ m\ n & = & m * n \uplus \mathit{rect}'\ (h_1\ m)\ (h_1\ n) \uplus \cdots \uplus \mathit{rect}'\ (h_1\ m)\ (h_n\ n) \\ & & \quad \uplus \cdots \\ & & \quad \uplus \mathit{rect}'\ (h_n\ m)\ (h_1\ n) \uplus \cdots \uplus \mathit{rect}'\ (h_n\ m)\ (h_n\ n) \end{array}$$

This representation requires more constructors than the first one ($n^2 + 1$ instead of $n + 1$). On the positive side, it can be easily generalized to higher dimensions.

5 Related and future work

This work is inspired by a recent paper of C. Okasaki [14], who derives representations of square matrices from exponentiation algorithms. He shows, in particular, that the tail-recursive version of the fast exponentiation gives rise to an implementation based on rightist right-perfect trees. Interestingly, the simpler

implementation based on fork-node trees is not mentioned. The reason is probably that fast exponentiation algorithms typically process the bits from least to most significant bit while fork-node trees and Braun trees are based on the reverse order. The relationship between number systems and data structures is explained at great length in [13]. The development in Section 3 can be seen as putting this design principle on a formal basis.

Extensions to the Hindley-Milner type system that allow to capture structural invariants in a more straightforward way have been described by C. Zenger [18, 19] and H. Xi [17]—the latter paper also appears in the proceedings of this workshop. Using the *indexed types* of C. Zenger one can, for instance, parameterize vectors and matrices by their size. Size compatibility is then statically ensured by the type checker. H. Xi achieves the same effect using dependent datatypes. In his system, *de Caml*, the type of perfect leaf trees is, for instance, declared as follows.

$$
\begin{array}{l}
\textbf{datatype}\ `a\ perfect\ \textbf{with}\ nat \\
\quad = \quad Leaf\ (0)\ \textbf{of}\ `a \\
\quad \mid \quad \{n : nat\} Fork\ (n+1)\ \textbf{of}\ `a\ perfect\ (n) * `a\ perfect\ (n)
\end{array}
$$

This definition is essentially a transliteration of the top-down definition of perfect leaf trees given in the introduction. A practical advantage of dependent types is that standard regular datatypes and functions on these types can be adapted with little or no change. Often it suffices to annotate datatype declarations and type signatures with appropriate size constraints.

Directions for future work suggest themselves. It remains to adapt the standard vector and matrix algorithms to the new representations. Some preparatory work has been done in this respect. In [9] the author shows how to adapt search tree algorithms to nested representations of search trees using constructor classes. It is conceivable that this approach can be applied to matrix algorithms, as well. Furthermore, many functions like *map*, *listify*, *sum* etc can be generated automatically using the technique of polytypic programming [11]. On the theoretical side, it would be interesting to investigate the expressiveness of the framework and of higher-order polymorphic types in general. Which class of multisets can be described using higher-order recursion equations? For instance, it appears to be impossible to specify the multisets of all prime numbers. Do higher-order kinds increase the expressiveness?

Acknowledgements

I am grateful to Iliano Cervesato, Lambert Meertens, and John O'Donnell for many valuable comments.

References

[1] G.M. Adel'son-Vel'skiĭ and Y.M. Landis. An algorithm for the organization of information. *Doklady Akademiia Nauk SSSR*, 146:263–266, 1962. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[3] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall Europe, London, 1997.

[4] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.

[5] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology, 1983.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1991.

[7] Victor J. Dielissen and Anne Kaldewaij. A simple, efficient, and flexible implementation of flexible arrays. In *Third International Conference on Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany*, volume 947 of *Lecture Notes in Computer Science*, pages 232–241. Springer-Verlag, July 1995.

[8] Leo J. Guibas and Robert Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[9] Ralf Hinze. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, December 1998.

[10] Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 1999. To appear.

[11] Ralf Hinze. Polytypic functions over nested datatypes (extended abstract). In *3rd Latin-American Conference on Functional Programming (CLaPF'99)*, March 1999.

[12] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[14] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, Paris, France*, 1999. To appear.

[15] Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999.

[16] Lawrence Snyder. On uniquely represented data structures (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence*, pages 142–146, Long Beach, Ca., USA, October 1977. IEEE Computer Society Press.

[17] Hongwei Xi. Dependently typed data structures. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France*, September 1999.

[18] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, November 1997.

[19] Christoph Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, 1998.

Dependently Typed Data Structures*

Hongwei Xi

Oregon Graduate Institute

Abstract

The mechanism for declaring datatypes in functional programming languages such as ML and Haskell is of great use in practice. This mechanism, however, often suffers from its imprecision in capturing the invariants inherent in data structures. We remedy the situation with the introduction of dependent datatypes so that we can model data structures with significantly more accuracy. We present a few interesting examples such as implementations of red-black trees and binomial heaps to illustrate the use of dependent datatypes in capturing some sophisticated invariants in data structures. We claim that dependent datatypes can enable the programmer to implement algorithms in a way that is more robust and easier to understand.

1 Introduction

The mechanism that allows the programmer to declare datatypes seems indispensable in functional programming languages such as Standard ML [15] and Haskell [19]. In practice, we often encounter situations where the declared datatypes do not accurately capture what we really need. For instance, if what we need is a data structure for the pairs of integer lists of the same length, we often declare a datatype in Standard ML or Haskell that is for *all* pairs of integer lists. This inaccuracy problem is often a rich source for program errors. A typical scenario is that a function which should only receive as its argument a pair of integer lists of the same length is mistakenly applied to a pair of integer lists of different lengths. Unfortunately, such a mistake causes no type errors if pairs of integer lists of equal length are given a type that is for all pairs of integer lists, and thus can usually hide in a program unnoticed until at run-time, when debugging often becomes much more demanding than at compile-time.

The inaccuracy problem becomes more serious when we start to implement more sophisticated data structures such as red-black trees, binomial heaps, ordered lists, etc. There are some relatively complex invariants in these data structures that we must maintain in order to implement them correctly. For instance, a correct implementation of an insertion operation on a red-black tree should always yield a red-black tree. If we can form a datatype to precisely capture the properties of a red-black tree, then it becomes possible to detect a program error through type-checking when such an error leads to the violation of one of these captured properties. This is evidently a desirable feature in programming if it can be made practical.

The need for forming more accurate datatypes partially motivated the design of Dependent ML (DML), an enrichment of ML with a restricted form of dependent types. More precisely, DML is a language schema. Given a constraint domain C, DML(C) is the language in the schema where all type index expressions are drawn from C. Roughly speaking, a type index expression is simply a term that can be used to index a type. Type-checking in DML(C) can then be reduced to constraint satisfaction in C. In this paper, we restrict C to some integer domain and use the name DML for this particular DML(C). A variant of DML, *de Caml*, has been implemented on top of Caml-light [14]. This implementation essentially replaces the front-end of

* Partially supported by the United States Air Force Materiel Command (F19 628-96-C-0161) and the Department of Defense.

Caml-light with a dependent type-checker and keeps the back-end of Caml-light intact. It also modifies many library functions, assigning to them more accurate types.

An alternative approach to forming more accurate datatypes is to use nested datatypes [2]. For instance, a nested datatype exactly representing red-black trees can be readily formed. However, there exist various significant differences between DML-style dependent types and nested datatypes, which we will illustrate later.

We use `typewriter font` in this paper to represent code in de Caml, all of which have been verified in a prototype implementation. A significant consequence of the introduction of dependent types is the loss of the notion of principal types in DML. For instance, both of the following types can be assigned to an implementation in de Caml which zips two lists together.

```
'a list * 'b list -> ('a * 'b) list
{n:nat}'a list(n) * 'b list(n) -> ('a * 'b) list(n)
```

The first type has the usually meaning, while the second one implies that for every natural number n, the function yields a list with length n when given a pair of lists with length n. Notice that we use `'a list(n)` for the type of a list with length n in which every element is of type `'a`. If a dependent type is to be assigned to a function in DML, it is the responsibility of the programmer to annotate the function with such a dependent type. This is probably the most significant difference between the programming styles in ML and in DML. In practice, we observe that the type annotations in a typical DML program often constitutes less than 20% of the entire code. Since dependent type annotations can often lead to more accurate reports of type error messages and serve as informative program documentation, we feel that the DML programming style is acceptable from a practical point of view. We will provide some concrete examples for the reader to judge this claim, including implementations of red-black trees and binomial heaps. Both of these implementations are adopted from the corresponding ones in [17]. The implementations in de Caml have several advantages over the original ones. We have verified more invariants in the de Caml implementations. For instance, it is verified in the type system of de Caml that the function which merges two binomial heaps indeed yields a binomial heap. Also the type annotations in the implementations, which can be fully trusted since they are mechanically verified, offer some pedagogical values. We feel it is easier to understand the de Caml implementations because the reader can reason in the presence of these informative dependent types.

In this paper, it is neither possible nor necessary to formally present DML. Instead, we focus on presenting some concrete examples in the programming language *de Caml*, a variant of DML, as well as some intuitive explanation. We refer the interested reader to [22] for the formal development of DML, though we strongly believe that this is largely unnecessary for comprehending this paper.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of the types in DML. We then introduce de Caml in Section 3, presenting some of its main features and illustrating type-checking in de Caml with a short example. Some case studies are given in Section 4, including implementations of Braun trees, random-access lists, red-black trees and binomial heaps. Lastly, we discuss some related work and then conclude.

2 Types in Dependent ML

In this section, we present a brief explanation on the types in DML. The reader is encouraged to skip this section and read it later, though it could be helpful to gather some intuition before studying the concrete examples in Section 3.

Intuitively speaking, dependent types are types which depend on the values of language expressions. For instance, we may form a type $int(i)$ for each integer i to mean that every integer expression of this type must have value i, that is, $int(i)$ is a singleton type. Note that i is the expression on which this type depends. We use the name *type index expression* for such an expression. There are various compelling reasons, such as practical type-checking, for imposing restrictions on expressions which can be chosen as type index expressions. A

$$
\begin{array}{rrcl}
\text{index expressions} & i,j & ::= & a \mid c \mid i+j \mid i-j \mid i*j \mid i \div j \mid \min(i,j) \mid \max(i,j) \mid \mathrm{mod}(i,j) \\
\text{index propositions} & P & ::= & i<j \mid i \leq j \mid i \geq j \mid i>j \mid i=j \mid i \neq j \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \\
\text{index sorts} & \gamma & ::= & int \mid \{a:\gamma \mid P\} \mid \gamma_1 * \gamma_2 \\
\text{index contexts} & \phi & ::= & \cdot \mid \phi, a:\gamma \mid \phi, P
\end{array}
$$

Figure 1: The syntax for type index expressions

novelty in DML is to require that type index expressions be drawn only from a given constraint domain. For the purpose of this paper, we restrict type index expressions to integers. We present the syntax for type index expressions in Figure 1, where we use a for type index variables and c for a fixed integer. Note that the language for type index expressions is typed. We use *sorts* for the types in this language in order to avoid potential confusion. We use $\cdot$ for the empty index variable context and omit the standard sorting rules for this language. We also use certain transparent abbreviations, such as $0 \leq i < j$ which stands for $0 \leq i \wedge i < j$. The subset sort $\{a:\gamma \mid P\}$ stands for the sort for those elements of sort γ which satisfy the proposition P. For example, we use *nat* as an abbreviation for the subset sort $\{a:int \mid a \geq 0\}$.

Types in DML are formed as follows. We use α for type variables and δ for type constructors.

$$
\text{types} \quad \tau \quad ::= \quad \alpha \mid (\tau_1,\ldots,\tau_n)\delta(i) \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \Pi a:\gamma.\tau \mid \Sigma a:\gamma.\tau
$$

For instance, $list$ is a type constructor and $(int)list(n)$ stands for the type of an integer list of length n. $\Pi a:\gamma.\tau$ and $\Sigma a:\gamma.\tau$ form a universal dependent type and an existential dependent type, respectively. For instance, the universal dependent type $\Pi a:nat.(int)list(a) \to (int)list(a)$ captures the invariant of a function which, for every natural number a, returns an integer list of length a when given an integer list of length a. Also we can use the existential dependent type $\Sigma a:nat.(int)list(a)$ to mean an integer list of some unknown length. We demonstrate how a type constructor is declared in Section 3.

The typing rules for this language should be familiar from a dependently typed λ-calculus (such as the ones underlying Coq or NuPrl). The critical notion of *type conversion* uses the judgment $\phi \vdash \tau_1 \equiv \tau_2$ which is the congruent extension of equality on index expressions to arbitrary types:

$$
\frac{\phi \models \tau_1 \equiv \tau_1' \quad \cdots \quad \tau_n \equiv \tau_n' \quad \phi \models i \doteq i'}{\phi \vdash (\tau_1,\ldots,\tau_n)\delta(i) \equiv (\tau_1',\ldots,\tau_n')\delta(i')}
$$

$$
\frac{\phi \vdash \tau_1 \equiv \tau_1' \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'} \qquad \frac{\phi \vdash \tau_1' \equiv \tau_1 \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}
$$

$$
\frac{\phi, a:\gamma \vdash \tau \equiv \tau'}{\phi \vdash \Pi a:\gamma.\tau \equiv \Pi a:\gamma.\tau'} \qquad \frac{\phi, a:\gamma \vdash \tau \equiv \tau'}{\phi \vdash \Sigma a:\gamma.\tau \equiv \Sigma a:\gamma.\tau'}
$$

Notice that it is the application of these rules which generates constraints. For instance, the constraint $\phi \models (a+n)+1 \doteq m+n$ is generated in order to derive $\phi \vdash (int)list((a+n)+1) \equiv (int)list(m+n)$.

It is difficult to present more details given the space limitation. For those who are interested, we point out that the detailed formal development of DML can be found in [22].

3 Some Features in de Caml

In this section, we use examples to present some unique and significant features in de Caml, preparing for the case studies in Section 4.

The programmer often declares datatypes when programming in ML. For instance, the following datatype declaration defines a type constructor $list$.

```
type 'a list = nil | cons of 'a * 'a list
```

Roughly speaking, this declaration states that a polymorphic list is formed with two constructors `nil` and `cons`, whose types are `'a list` and `'a * 'a list -> 'a list`, respectively. We use `'a` for a type variable. However, the declared type `'a list` is coarse. For instance, we cannot use the type to distinguish an empty list from a non-empty one. In de Caml, this type can be refined as follows.

```
refine 'a list with nat =
  nil(0) | {n:nat} cons(n+1) 'a * 'a list(n)
```

The clause `refine 'a list with nat` means that we refine the type `'a list` with an index of sort `nat`, that is, the index is a natural number. In this case, the index stands for the length of a list.

- `nil(0)` means that `nil` is of type `'a list(0)`, that is, it is a list of length 0.
- `{n:nat} cons(n+1) of 'a * 'a list(n)` means that `cons` is of type

  ```
  {n:nat} 'a * 'a list(n) -> 'a list(n+1),
  ```

  that is, for every natural number `n`, `cons` yields a list of length $n + 1$ when given an element of type `'a` and a list of length n. Note `{n:nat}` is a universal quantifier, which is usually written as $\Pi n : nat$ in type theory.

Now list types have become more informative. The following code defines the append function on lists. We use `[]` for `nil` and `::` as the infix operator for `cons`.

```
let rec append = function
    ([], ys) -> ys
  | (x :: xs, ys) -> x :: append(xs, ys)
withtype {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

The `withtype` clause is a type annotation supplied by the programmer, which simply states that the function returns a list of length of $m + n$ when given a pair of lists of lengths m and n, respectively. We now present an informal description about type-checking in this case.

For the first clause `([], ys) -> ys`, the type-checker assumes that `ys` is of types `'a list(b)` for some index variable b of sort `nat`. This implies that `([], ys)` is of type `'a list(0) * 'a list(b)`. The type-checker then instantiates m and n with 0 and b, respectively, and verify that the `ys` on the right side of `->` is of type `'a list(0+b)`. Since `ys` is of type `'a list(b)` under assumption, the type-checker generates a constraint $b = 0 + b$ under the assumption that b is a natural number. This constraint can be easily verified.

Let us now type-check the second clause `(x :: xs, ys) -> x :: append(xs, ys)`. Assume that `xs` and `ys` are of type `'a list(a)` and `'a list(b)`, respectively, where a and b are index variables of sort `nat`. Then `(x :: xs, ys)` is of type `'a list(a+1) * 'a list(b)`, and we therefore instantiate m and n with $a + 1$ and b, respectively. Also we infer that the right side `x :: append(xs, ys)` is of type `'a list((a+b)+1)` since xs and ys are assumed of types `'a list(a)` and `'a list(b)`, respectively. We need to prove that the right side is of type `int list(m+n)` for $m = a + 1$ and $n = b$. This leads to the following constraint,

$$(a + 1) + b = (a + b) + 1$$

which can be immediately verified under that assumption that a and b are natural numbers. This finishes type-checking the above de Caml program. The interested reader is referred to [22] for the formal presentation of type-checking in DML.

Clearly, a natural question is whether the type for `append` can be reconstructed or synthesized. For such a simple example, this seems highly possible. However, our experience indicates that it seems exceedingly difficult in general to synthesize dependent types in practice, though we have not formally studied this issue.

Instead of refining a type, it is also allowed to declare a dependent type in de Caml. For instance, we can declare the following.

```
datatype 'a list with nat = nil(0) | {n:nat} cons of 'a * 'a list(n)
```

The declaration is basically equivalent to the refinement we made earlier. However, there is also a significant difference. When we declare a refinement, we must be able to interpret the corresponding unrefined types in terms of refined ones. For example, after refining the type `'a list`, we must interpret this type in terms of the refined list type. We need existential dependent types for this purpose. `'a list` is interpreted as `[n:nat] 'a list(n)`, that is, `'a list` is `'a list(n)` for some (unknown) natural number n. Note that `[n:nat]` is an existential quantifier, which is often written as $\Sigma n : nat$ in type theory. This provides a smooth interaction between ML types and dependent types. Suppose that *f* is defined before the list type is refined and its type is `'a list -> 'a list`. After refining the list type, we can assign to *f* the type `([n:nat] 'a list) -> [n:nat] 'a list`, that is, *f* takes a list with unknown length and returns a list with unknown length. This makes it possible for *f* to be applied to an argument of dependent type, say, `int list(2)`. This is also essential for ensuring backward compatibility, a very important issue when the use of existing ML code is concerned.

However, there is a need for imposing some restriction on datatype refinement. We give a short example to illustrate such a need. The datatype `'a tree` is declared as follows for *all* binary trees.

```
datatype 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Suppose we declare the following refinement, where the type index standards for the height of a tree.

```
refine 'a tree with nat =
  Leaf(0) | {h:nat} Node(h+1) of 'a tree(h) * 'a * 'a tree(h)
```

This refinement is problematic since the type `[h:nat] 'a tree(h)` now standards for the type of all *perfect* binary trees, and therefore it cannot be used to represent the original `'a tree`, which is the type for all binary trees. There is some syntactic restriction that can be imposed to rule out such problematic datatype refinements. We stop mentioning the restriction since it is simply not needed in this paper.

There is another important use of existential dependent types. In order to guarantee practical type checking in de Caml, we must make constraints relatively simple. Currently, we only accept linear integer constraints. This immediately implies that there are many (realistic) constraints that are inexpressible in the type system of de Caml. For instance, the following code implements a filter function on a list which removes from the list all elements not satisfying the property `p`.

```
let filter p = function
    [] -> []
  | x :: xs -> if p(x) then x :: (filter p xs) else (filter p xs)
```

In general, it is impossible to know the length of the list `(filter p l)` without knowing what `p` is. Therefore, it is impossible to type the function using only universal dependent types. Nonetheless, we know that the length of `(filter p xs)` is less than or equal to that of `l`. This invariant can be captured by assigning `filter` the following types.

```
('a -> bool) -> {m:nat} 'a list(m) -> [n:nat | n <= m] 'a list(n)
```

Note that `[n:nat | n <= m]` stands for $\Sigma n : \{a : nat \mid a \leq m\}$.

Another significant use of existential dependent types is to represent a range of values. We can use `([n:nat] int(n)) array` to represent the type for the vectors whose elements are natural numbers.

```
datatype 'a brauntree with nat =
    L(0)
  | {m:nat}{n:nat |  n <= m <= n+1}
    B(m+n+1) of 'a * 'a brauntree(m) * 'a brauntree(n) ;;

let rec diff k = function
    L -> 0
  | B(_, l, r) ->
    if k = 0 then 1
    else if k mod 2 = 1 then diff (k/2) l else diff (k/2 - 1) r
withtype {k:nat}{n:nat | k <= n <= k+1}
         int(k) -> 'a brauntree(n) -> int(n-k) ;;

let rec size = function
    L -> 0 | B(_, l, r) -> let n = size r in 1 + n + n + diff n l
withtype {n:nat} 'a brauntree(n) -> int(n) ;;
```

Figure 2: An implementation of the size function on Braun trees

This is very useful for eliminating array bound checks at run-time [20]. In general, we view that the use of existential types in de Caml for handling functions like `filter` is crucial to the scalability of the type system of de Caml since such functions are abundant in practice.

Lastly, we mention a convention in de Caml. After declaring a dependent type as follows,

$$\texttt{datatype } (\alpha_1, \ldots, \alpha_m)\, \delta \texttt{ with } (sort_1, \ldots, sort_n) = \cdots\cdots$$

we may write $(\tau_1, \ldots, \tau_m)\delta$ to stand for the following.

$$[a_1 : sort_1] \cdots [a_n : sort_n].(\tau_1, \ldots, \tau_m)\delta(a_1, \ldots, a_n)$$

For example, `'a list` stands for `[n:nat] 'a list(n)`.

4 Case Studies

In this section, we present some examples to demonstrate the use of dependent datatypes in capturing invariants in data structures. All these examples in de Caml have been successfully verified in a prototype compiler for de Caml, which is written on top of the Caml-light compiler [14]. The claim we make is that dependent datatypes enable the programmer to implement algorithms in a way that is more robust and easier to understand.

4.1 Braun Trees

A Braun tree is a balanced binary tree [4] such that for every branch node in the tree, its left subtree is either the same size as its right subtree, or contains one more element. Braun trees can be used to give neat implementations for flexible arrays and priority queues. In [16], there is an algorithm which computes the size of a Braun tree in $O(\log^2 n)$ time, where n is the size of the Braun tree. We implement this algorithm in Figure 2. We first declare a dependent datatype `'a brauntree(n)` for Braun trees of size n. Note that the type of `B` is

```
{m:nat}{n:nat | n <= m <= n+1}
'a * 'a brauntree(m) * 'a brauntree(n) -> 'a brauntree(m+n+1)
```

which states that `B` yields a Braun tree of size $m + n + 1$ when given an element, a Braun tree of size m and a Braun tree of size n where $n \leq m \leq n + 1$ holds. This exactly captures the invariant on Braun trees mentioned above.

Given a natural number k and a Braun tree of size n satisfying $k \leq n \leq k + 1$, the function `diff` yields the difference between n and k. With this function, the size function on Braun trees can be defined straightforwardly. An interesting point in this example is that the type of the function `size` precisely indicates that this is the size function on Braun trees since it states that the function returns an integer of value n when given a Braun tree of size n.

The reason that `diff` n l yields the difference between $|l|$, the size of l, and n can be found in [16]. We give some brief explanation below. It is clear that $|l| - n$ is either 0 or 1. If l is a leaf, $|l| - n$ must be 0. Otherwise, $|l| = 1 + |l'| + |r'|$, where l' and r' are the left and right branches of l, respectively. If n is odd, then $n = 1 + \lfloor n/2 \rfloor + \lfloor n/2 \rfloor$ and thus

$$|l| - n \quad = \quad 1 + |l'| + |r'| - 1 + \lfloor n/2 \rfloor + \lfloor n/2 \rfloor = (|l'| - \lfloor n/2 \rfloor) + (|r'| - \lfloor n/2 \rfloor)$$

Since $|l'| - 1 \leq |r'| \leq |l'|$ holds, we have the following.

$$2(|l'| - \lfloor n/2 \rfloor) - 1 \leq |l| - n \leq 2(|l'| - \lfloor n/2 \rfloor)$$

It can now be readily verified that $|l| - n = 1$ if $|l'| - \lfloor n/2 \rfloor = 1$ and $|l| - n = 0$ if $|l'| - \lfloor n/2 \rfloor = 0$. Therefore, if n is odd, $|l| - n = |l'| - \lfloor n/2 \rfloor$. With some similar reasoning, we can eventually prove the correctness of the defined function `diff`.

This example also shows that although the datatype type declaration for Braun trees contains size information, this information is not available at run-time and therefore a recursive walk through the tree is necessary to determine the size of a tree.

4.2 Random-Access Lists

A random-access list is a list representation such that list lookup (update) can be implemented in an efficient way. In this case, the lookup (update) function takes $O(\log n)$ time in contrast to the usual $O(n)$ time (worst case), where n is the length of the input list.

We present an implementation of random-access list in Figures 3 and 4. We first declare the dependent datatype for random-access lists. Note that `'a rlist(n)` stands for the type of random-access lists with length n. `Nil` and `One` are the constructors for empty and singleton random-access lists, respectively. Furthermore, the constructors `Even` and `Odd` are to form random-access lists of even and odd lengths, respectively. If `l1` and `l2` represent lists $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ for some $n > 0$, respectively, then `Even (l1, l2)` represents the list $x_1, y_1, \ldots, x_n, y_n$. Similarly, if `l1` and `l2` represent lists $x_1, \ldots, x_n, x_{n+1}$ and $y_1, \ldots, y_n$ for some $n > 0$, respectively, `Odd(l1, l2)` represents $x_1, y_1, \ldots, x_n, y_n, x_{n+1}$. With such a data structure, we can implement a lookup (update) function on random-access list which takes $O(\log n)$ time. A crucial invariant on this data structure is that `l1` and `l2` must have the same length if `Even(l1, l2)` is formed or `l1` contains one more element than `l2` if `Odd(l1, l2)` is formed. This is clearly captured by the dependent datatype declaration for `'a rlist`. The function `cons` appends an element to a list and `uncons` decomposes a list into a pair consisting of the head and the tail of the list. Note that the type of `uncons` requires this function only to be applied to a non-empty list. Both `cons` and `uncons` takes $O(\log n)$ time.

The function `lookup_safe` deserves some explanation. The type of this function indicates that it can be applied to `i` and `l` only if `i` is a natural number and its value is less than the length of `l`. Notice that the `look_up i l` simply return `x` when the `l` matches the pattern `One x`. There is no need to check whether

```
datatype 'a rlist with nat =
    Nil(0)
  | One(1) of 'a
  | {n:nat | n > 0} Even(n+n) of 'a rlist(n) * 'a rlist(n)
  | {n:nat | n > 0} Odd(n+n+1) of 'a rlist(n+1) * 'a rlist(n) ;;

exception Subscript ;;

let rec cons x = function
    Nil -> One x
  | One y ->  Even(One(x), One(y))
  | Even(l1, l2) -> Odd(cons x l2, l1)
  | Odd(l1, l2) -> Even(cons x l2, l1)
withtype {n:nat} 'a -> 'a rlist(n) -> 'a rlist(n+1) ;;

let rec uncons = function
    One x -> (x, Nil)
  | Even(l1, l2) ->
    let (x, l1) = uncons l1 in begin
      match l1 with
        Nil -> (x, l2) | _ -> (x, Odd(l2, l1))
    end
  | Odd(l1, l2) -> let (x, l1) = uncons l1 in (x, Even(l2, l1))
withtype {n:nat | n > 0} 'a rlist(n) -> 'a * 'a rlist(n-1) ;;

let rec length = function
    Nil -> 0
  | One _ -> 1
  | Even (l1, _) -> 2 * (length l1)
  | Odd (_, l2) -> 2 * (length l2) + 1
withtype {n:nat} 'a rlist(n) -> int(n) ;;

let rec lookup_safe i = function
    One x -> x
  | Even (l1, l2) ->
    if i mod 2 = 0 then lookup_safe (i / 2) l1
    else lookup_safe (i / 2) l2
  | Odd(l1, l2) ->
    if i mod 2 = 0 then lookup_safe (i / 2) l1
    else lookup_safe (i / 2) l2
withtype {i:nat}{n:nat | i < n} int(i) -> 'a rlist(n) -> 'a ;;
```

Figure 3: An implementation of random-access lists in de Caml (I)

```
let rec update_safe i x = function
    One y -> One x
  | Even(l1, l2) ->
    if i mod 2 = 0 then Even(update_safe (i / 2) x l1, l2)
    else Even(l1, update_safe (i / 2) x l2)
  | Odd(l1, l2) ->
    if i mod 2 = 0 then Odd(update_safe (i / 2) x l1, l2)
    else Odd(l1, update_safe (i / 2) x l2)
withtype {i:nat}{n:nat | i < n}
         int(i) -> 'a -> 'a rlist(n) -> 'a rlist(n) ;;
```

Figure 4: An implementation of random-access lists in de Caml (II)

```
datatype 'a rlist with nat =
    Nil(0)
  | One(1) of 'a
  | {n:nat | n > 0} Even(n+n) of ('a * 'a) rlist(n)
  | {n:nat | n > 0} Odd(n+n+1) of 'a * ('a * 'a) rlist(n)
```

Figure 5: A nested dependent datatype for random access lists

`i` is 0: it must be since `i` is a natural number and `i` is less than the length of `l`, which is 1 in this case. The usual lookup function can be implemented as usual or as follows.

```
let rec lookup i l =
  if i < 0 then raise Subscript
  else if i >= length l then raise Subscript
       else lookup_safe i l
withtype int -> 'a rlist -> 'a ;;
```

We point out that an implementation of random-access lists is given in [17], which uses the feature of nested datatypes. Okasaki's implementation supports (on average) $O(1)$-time consing and unconsing operations and are thus superior to our implementation in this respect. On the other hand, the update function in Okasaki's implementation requires the use of some higher-order feature, which does not exist in our implementation. We view this as an edge of our implementation.

It should be stressed that nested datatypes and DML-style dependent types are orthogonal to each other. For instance, we can form a nested dependent datatype in Figure 5 for random-access lists, imitating a corresponding datatype in [17]. Unfortunately, we currently cannot experiment with such a dependent datatype because polymorphic recursion is not supported in Caml-light.

4.3 Red-Black Trees

A red-black tree (RBT) is a balanced binary tree which satisfies the following conditions: (a) all leaves are marked black and all other nodes are marked either red or black; (b) for every node there are the same number of black nodes on every path connecting the node to a leaf, and this number is called the *black height* of the node; (c) the two sons of every red node must be black. It is a common practice to use the RBT data structure for implementing a dictionary. We declare a datatype in Figure 6, which precisely captures these properties of a RBT.

```
type key == int ;;

sort color == {a:int | 0 <= a <= 1} ;;

datatype rbtree with (color, nat, nat) =
    E(0, 0, 0)
  | {c:color}{cl:color}{cr:color}{bh:nat}
    B(0, bh+1, 0) of rbtree(cl, bh, 0) * key * rbtree(cr, bh, 0)
  | {cl:color}{cr:color}{bh:nat}
    R(1, bh, cl+cr) of rbtree(cl, bh, 0) * key * rbtree(cr, bh, 0) ;;

let restore = function
    (R(R(a, x, b), y, c), z, d) -> R(B(a, x, b), y, B(c, z, d))
  | (R(a, x, R(b, y, c)), z, d) -> R(B(a, x, b), y, B(c, z, d))
  | (a, x, R(R(b, y, c), z, d)) -> R(B(a, x, b), y, B(c, z, d))
  | (a, x, R(b, y, R(c, z, d))) -> R(B(a, x, b), y, B(c, z, d))
  | (a, x, b) -> B(a, x, b)
withtype {cl:color}{cr:color}{bh:nat}{vl:nat}{vr:nat | vl+vr <= 1}
         rbtree(cl, bh, vl) * key * rbtree(cr, bh, vr) ->
         [c:color] rbtree(c, bh+1, 0) ;;

exception Item_already_exists ;;

let insert x t =
  let rec ins = function
      E -> R(E, x, E)
    | B(a, y, b) -> if x < y then restore(ins a, y, b)
                    else if y < x then restore(a, y, ins b)
                         else raise Item_already_exists
    | R(a, y, b) -> if x < y then R(ins a, y, b)
                    else if y < x then R(a, y, ins b)
                         else raise Item_already_exists
  withtype {c:color}{bh:nat}
           rbtree(c, bh, 0) ->
           [c':color][v:nat | v <= c] rbtree(c', bh, v) in
  match ins t with
    R(a, y, b) -> B(a, y, b)
  | t -> t
withtype {c:color}{bh:nat} key -> rbtree(c, bh, 0) ->
         [bh':nat] rbtree(0, bh', 0) ;;
```

Figure 6: A red-black tree implementation

A sort `color` is declared for the type index expressions representing the colors of nodes. We use 0 for black and 1 for red. For simplicity, we use integers for keys. Of course, one can readily use other ordered data structures. The type `rbtree` is indexed with a triple `(c, bh, v)`, where c is the color of the node, bh is the black height of the tree, and v is the number of color violations. We record one color violation if a red node is followed by another red node, and thus a RBT must have no color violations. Clearly, the types of constructors indicate that color violations cano only occur at the top node. Also, notice that a leaf, that is, `E`, is considered black. Given the datatype declaration and the explanation, it should be clear that the type of a RBT is simply

```
[c:color][bh:nat] rbtree(c,bh,0),
```

that is, a tree which has some top node color c and some black height bh but no color violations.

It is an involved task to implement RBT. The implementation we present is basically adopted from the one in [17], though there are some minor modifications. We explain how the insertion operation on a RBT is implemented. Clearly, the invariant we intend to capture is that inserting an entry into a RBT yields another RBT. In other words, we intend to declare that the insertion operation is of the following type.

```
key->[c:color][bh:nat] rbtree(c,bh,0) -> [c:color][bh:nat] rbtree(c,bh,0)
```

If we insert an entry into a RBT, some properties on RBT may be violated. These properties can be restored through some rotation operations. The function `restore` in Figure 6 is defined for this purpose.

The type of `restore` is easy to understand. It states that this function takes an entry, a tree with at most one color violation and a RBT and returns a RBT tree. The two trees in the argument must have the same black height bh for some natural number bh and the returned RBT has black height $bh + 1$. This information can be of great help for understanding the code. If the information had been informally expressed through comments, it would be difficult to know whether the comments can be trusted. Also notice that it is not trivial at all to verify the information manually. We could imagine that almost everyone who did this would appreciate the availability of a type-checker to perform it automatically.

There is a great difference between type-checking a pattern matching clauses in DML and in ML. The operational semantics of ML requires that pattern matching be performed sequentially, that is, the chosen pattern matching clause is always the first one which matches a given value. For instance, in the definition of the function `restore`, if the last clause is chosen at run-time, then we know the argument of `restore` does not match either of the clauses ahead of the last one. This must be taken into account when we type-checking pattern matching in DML. One approach is to expand patterns into disjoint ones. For instance, the pattern `(a, x, b)` expands into 36 patterns $(pattern_1, x, pattern_2)$, where $pattern_1$ and $pattern_2$ range over the following six patterns: `R(B _, _, B _)`, `R(B _, _, E)`, `R(E, _, B _)`, `R(E, _, E)`, `B _`, and `E`. Unfortunately, such expansion may lead to combinatorial explosion. An alternative is to require the programmer to indicate whether such expansion is needed. Neither of these is currently available in de Caml, and the author has taken the inconvenience to expand patterns into disjoint ones when necessary. We emphasize that the code in Figure 6 must be thus expanded in order to pass type-checking in de Caml. Though this can be fixed straightforwardly, it is currently unclear what method can solve the problem best.

The complete implementation of the insertion operation follows immediately. Notice that the type of function `ins` indicates that `ins` may return a tree with one color violation if it is applied to a tree with red top node. This is fixed by replacing the top node with a black one for every returned tree with a red top node.

Moreover, we can use an extra index to indicate the size of a RBT. If we do so, we can then show that the `insert` function always returns a RBT of size $n + 1$ when given a RBT of size n (note that an exception is raised if the inserted entry already exists in the tree). Please refer to [23] for details.

4.4 Binomial Heaps

A binomial tree is defined recursively; a binomial tree B_0 with rank 0 consists of a single node and a binomial tree B_{k+1} of rank $k + 1$ consists of two linked binomial trees B_k of rank k such that the root of one B_k is the

leftmost son of the other B_k. A binomial heap H is a collection of binomial trees that satisfy the properties: (a) each binomial tree in H is heap-ordered, that is, the key of a node is greater that or equal to the key of its parent, and (b) there is at most one binomial tree in H whose root has a given degree. Please refer to [6] for details.

We declare some datatypes in Figure 7 for forming binomial heaps. The type `tree(n)` is for binomial trees of rank n, and the type `treelist(n)` is for a list of binomial trees with *decreasing* ranks and $n = m+1$ if the list is not empty, where m is the rank of the first binomial tree in the list. We represent a binomial heap as a list of binomial trees with *increasing* ranks. For a heap of type `heap(n)`, if $n = 0$ then the heap is empty; otherwise $n = m + 1$ where m is the rank of the first binomial tree in the heap. Notice that we attach rank to each tree node in order to efficiently compute the rank of a tree while using the type of `Node` to guarantee that the first component of each node indeed represents the rank of that node.

Notice that the datatype for binomial trees *does not* capture the invariant stating that these trees are heap-ordered. This seems to be beyond the reach of dependent datatypes. Also note that we would not be able to capture some of the invariants if we used the ordinary list constructors, that is, `nil` and `cons`, to form tree lists. This leads to the introduction of `Tempty`, `Tcons`, `Hempty` and `Hcons`. This special feature in programming with dependent datatypes has an unpleasant consequence, which we mention in Section 5.

The implementation in Figure 7 and 8 is largely adopted from [17]. Since the type for the function `merge` is relatively complex, we explain it as follows. This type states that given two binomial heaps of types `heap(m)` and `heap(n)`, respectively, this function returns a binomial heap of type `heap(l)` for some l such that $l = m$ if $n = 0$, or $l = n$ if $m = 0$, or $l \geq \min(m, n) > 0$ otherwise.

5 Limitation

We mention some limitations of dependent datatypes in this section.

In order to capture invariants, we may have to declare new datatypes instead of using existing ones. For instance, we declared the datatype `treelist` in Figure 7 instead of using the existing list constructors to form a list of trees. The reason is that we wanted to only form lists of binomial trees with decreasing rank. Similarly, we introduced the datatype `heap` to capture the invariant that a binomial heap is a list of trees with increasing order. This forces us to define the function `to_heap` later, which essentially reverses a list of trees and append it to a heap. If we used the existing list constructors without declaring either of `treelist` and `heap`, we could then use some existing function on lists instead of defining `to_heap`. In order words, using dependent datatypes may lose some opportunities for code reuse.

Another limitation can be illustrated using the following example. Let `B` be the constructor declared in Figure 2, which is used to form Braun trees. Suppose that `B(x, l, r)` occurs in the code where *the programmer knows* for some reason that `l` is the same size as `r` or contains one more element but this cannot be established in the type system of de Caml. In this case, the code is to be rejected by the de Caml type-checker, though the code will cause no run-time error (if we trust the programmer). The situation is very similar to the case where we move from an untyped programming language into a typed one. A solution to this problem is that we introduce some run-time checks. For instance, we may define the following function and replace `B(x, l, r)` with `make_brauntree x l r`.

```
let make_brauntree x l r =
  let m = size(l) and n = size(r) in
    if n <= m && m <= n+1 then B(x, l, r) else raise Illegal_argument
withtype int -> brauntree -> brauntree -> brauntree
```

The function `make_brauntree` can readily pass type-checking in de Caml (we refer the interested reader to [22] for further details). The penalty in this case is that `make_brauntree` takes $O(\log^2 n)$ time to build a tree of size n, though this can be avoided if we store size information in each node.

In general, if the programmer anticipates the above situation to occur frequently, then she or he should either make sure that run-time checks can be done efficiently or switch back to non-dependent datatypes.

```
datatype tree with nat =
    {n:nat} Node(n) of int(n) * int * treelist(n)

and treelist with nat =
    Tempty(0)
  | {m:nat}{n:nat | m >= n} Tcons(m+1) of tree(m) * treelist(n) ;;

datatype heap with nat =
    Hempty(0)
  | {m:nat}{n:nat | n = 0 \/ m+1 < n} Hcons(m+1) of tree(m) * heap(n) ;;

let rank = function Node(r, _, _) -> r
withtype {n:nat} tree(n) -> int(n) ;;

let root = function Node(_, x, _) -> x
withtype {n:nat} tree(n) -> int ;;

let link (Node(r, x1, ts1) as t1) = function
    Node(_, x2, ts2) as t2 ->
    if (x1 <= x2) then Node(r+1, x1, Tcons(t2, ts1))
    else Node(r+1, x2, Tcons(t1, ts2))
withtype {r:nat} tree(r) -> tree(r) -> tree(r+1) ;;

let rec insTree t = function
    Hempty -> Hcons(t, Hempty)
  | Hcons(t', ts') as ts ->
    if rank t < rank t' then Hcons(t, ts) else insTree (link t t') ts'
withtype {r:nat}{n:nat | n = 0 \/ r < n}
         tree(r) -> heap(n) -> [l:nat | l > r] heap(l) ;;

let insert x hp = insTree (Node(0, x, Tempty)) hp
withtype int -> [n:nat] heap(n) -> [n:nat | n > 0] heap(n) ;;

let rec merge = function
    (hp1, Hempty) -> hp1
  | (Hempty, hp2) -> hp2
  | (Hcons(t1, hp1') as hp1), (Hcons(t2, hp2') as hp2) ->
    if rank t1 < rank t2 then Hcons(t1, merge(hp1', hp2))
    else if rank t1 > rank t2 then Hcons(t2, merge(hp1, hp2'))
    else let hp = merge(hp1', hp2') in insTree (link t1 t2) hp
withtype {m:nat}{n:nat} heap(m) * heap(n) ->
         [l:nat | (n = 0 /\ l = m) \/ (m = 0 /\ l = n) \/
                  (l >= min(m, n) > 0)] heap(l) ;;
```

Figure 7: An implementation of binomial heap in de Caml (I)

```
exception Heap_is_empty ;;

let rec removeMinTree = function
    Hempty -> raise Heap_is_empty
  | Hcons(t, Hempty) -> (t, Hempty)
  | Hcons(t, hp) ->
    let (t', hp') = removeMinTree hp in
    if root t < root t' then (t, hp) else (t', Hcons(t, hp'))
withtype {n:nat}
         heap(n) ->
         [r:nat][l:nat | l = 0 \/ l >= n > 0] (tree(r) * heap(l)) ;;

let findMin hp = let (t, _) = removeMinTree hp in root t
withtype {n:nat} heap(n) -> int ;;

let rec to_heap hp = function
    Tempty -> hp
  | Tcons(t, ts) -> to_heap (Hcons(t, hp)) ts
withtype {m:nat}{n:nat | m = 0 \/ m > n}
         heap(m) -> treelist(n) -> heap ;;

let deleteMin hp =
  let (Node(_, x, ts), hp) = removeMinTree hp
  in merge (to_heap Hempty ts, hp)
withtype heap -> heap ;;
```

Figure 8: An implementation of binomial heap in de Caml (II)

We recommend that the programmer avoid complex encodings when using dependent datatypes to capture invariants in data structures.

6 Related Work

The use of type systems in program error detection is ubiquitous. Usually, the types in general purpose programming languages such as ML and Java are relatively inexpressive for the sake of practical type-checking. In these languages, the use of types in program verification is effective but too limited. Our work can be viewed as providing a more expressive type system to allow the programmer to capture more program properties through types and thus catch more errors at compile-time. As a consequence, types can serve as informative program documentation, facilitating program comprehension. We assign priority to the practicality of type-checking in our language design and emphasize the need for restricting the expressiveness of a type system.

In [21], we have compared our work with some traditional dependent type systems such as the ones underlining Coq [8] and NuPrl [5], which are far more refined than the type system of DML. There, we also give comparison to the notion of indexed types [25] (an earlier version of which is described in [24]), the notion of refinement types [9, 7], the notion of sized types [13], and the programming language Cayenne [1].

There have been many recent studies on the use of nested datatypes [2] in constructing (sophisticated) datatypes to capture more invariants in data structures. For instance, a variety of examples can be found in

[3, 18, 10, 12, 11]. We feel that the advantage of this approach is that it requires relatively minor language extensions, which may include polymorphic recursion, higher-order kinds, rank-2 polymorphism, to existing functional programming languages such as Haskell, while type-checking in DML is much more involved. On the other hand, this approach seems less flexible, often requiring some involved treatment at both type and program level. The important notion of datatype refinement in DML cannot be captured with nested datatypes. For instance, it is impossible to form a nested datatype that can capture the notion of the length of a list since this would imply that one could simply use types to distinguish non-empty lists from empty ones. In general, we think that these two approaches are essentially orthogonal in spite of some similar motivations behind their development and they can be readily combined with little effort.

7 Conclusion

The use of dependent datatypes in capturing invariants in data structures is novel. This practice can offer many advantages when we implement algorithms in advanced programming languages equipped with such a mechanism. The most significant advantage is probably in program error detection. We argued in Section 1 that the imprecision of datatypes in Standard ML or Haskell in capturing invariants can be a rich source for run-time program errors. In addition, the dependent type annotations supplied by the programmer are mechanically verified and can thus be fully trusted. They can serve as valuable program documentation, facilitating program understanding. There are also various uses of dependent datatypes in compiler optimization.

Type-checking in DML is largely independent of the size of a program since a type-checking unit is roughly the body of a toplevel function. In general, what matters in type-checking is the difficulty level of the properties that are to be checked. A more serious issue is how to report error messages in case of type errors. The type-checking in de Caml implements a top-down style algorithm, which usually pinpoints to the location of a type error. Unfortunately, the author finds that it may often be surprisingly difficult to figure out the cause of a type error. On the positive side, the type-checker of de Caml is often capable of detecting a variety of subtle errors. For instance, the author once used `Even(l1, l2)` to form a random-list (in Figure 3) and the type-checker raised an error because it could not prove that `l1` cannot be `Nil`. If this had gone unnoticed, it would have invalidated some invariant assumed by the programmer, potentially causing (difficult) run-time errors. We are currently in the process of gathering more statistics regarding the use of de Caml.

The usual focus of data structure design is mainly on enhancing time and/or space efficiency, and less attention is paid to program error detection. The introduction of dependent datatypes provides an opportunity to remedy the situation. In general, we are interested in promoting the use of light-weight formal methods in practical programming, enhancing the robustness of programs. We have presented some concrete examples of dependent datatypes in this paper in support of such a promotion. We hope these examples can raise the awareness of dependent datatypes and their use in implementing algorithms.

8 Acknowledgment

I thank Chris Okasaki, Ralf Hinze and an anonymous referee for their constructive comments, which have undoubtedly raised the quality of the paper. Also I thank Jim Hook for his generous help in presenting the paper.

References

[1] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[2] Richard Bird and Lambert Meertens. Nested datatypes. In *Mathematics of program construction*, pages 52–67. Springer-Verlag LNCS 1422, 1998.

[3] Richard Bird and Ross Paterson. de bruijn notation as a nested datatypes. *Journal of Functional Programming*, To appear.

[4] W. Braun and M. Rem. A logarithmic implementation of fexible arrays. Technical Report Memorandum MS83/1, Eidenhoven University of Technology, 1983.

[5] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[6] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1989.

[7] Rowan Davies. Practical refinement-type checking. Thesis Proposal, November 1997.

[8] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[9] Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, 1991.

[10] Ralf Hinze. Numerical Representations as Higher-Order Nested Types. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, April 1998.

[11] Ralf Hinze. Constructing Red-Black Trees. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, September 1999.

[12] Ralf Hinze. Manufacturing Datatypes. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, September 1999. Also available as Technical Report IAI-TR-99-5, Institut für Informatik III, Universität Bonn.

[13] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[14] INRIA. Caml-light. `http://caml.inria.fr`.

[15] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.

[16] Chris Okasaki. Three Algorithms on Braun Trees by Chris Okasaki. *Journal of Functional Programming*, 7(6):661–666, November 1997.

[17] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[18] Chris Okasaki. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, September 1999.

[19] Simon Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available from `http://www.haskell.org/onlinereport/`, February 1999.

[20] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[22] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[23] Hongwei Xi. Some programming examples in de Caml. Available at `http://www.cse.ogi.edu/~hongwei/DML/deCaml/examples/`, 1999.

[24] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

[25] Christoph Zenger. *Indizierte Typen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1998. Forthcoming.

Teaching monadic algorithms to first-year students*

Ricardo Peña Yolanda Ortega
Fernando Rubio

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, 28040 Madrid, Spain.
e-mails: {ricardo, yolanda}@sip.ucm.es, rubiod@eucmax.sim.ucm.es

Abstract

The main claim of this paper is that imperative concepts such as sequencing, repetition, mutable state, and I/O can be taught to first-year students by using the monadic facilities of a functional language such as Haskell.

We report on an experience of teaching algorithms involving arrays, and which are typical of a first programming course —such as *insertion sort*, *bubble sort*, *linear search*, and so on—, by using the monadic style. It appears that our students do not have special difficulties in grasping both the imperative concepts and the algorithms. They learn these algorithms after a previous exposition to classical functional programming.

In the paper, we provide a rich sample of the algorithms used in the course. We also claim that higher order constructions facilitate to our students the design of complex monadic algorithms.

Keywords: imperative functional programming, monadic algorithms, education.

1 Introduction

Since the end of the 80's, there has been a broad trend to abandon imperative languages on behalf of functional ones in introductory programming courses. So, at many universities, Pascal has been replaced by Scheme, ML —and its variants—, or Miranda, or, more recently, Gofer and Haskell, as the first programming language to be learnt by undergraduates. This kind of experiences have been already reported in a number of papers (see, for instance, [7, 9, 8]). Therefore, there is no need to repeat here the benefits of the functional paradigm for 'unexperienced' students.

Being the weak-point of functional programming languages execution efficiency, most of the recent research on the functional field has been devoted to increase the efficiency of functional programs. One of the most interesting results is *monadic programming* [15, 17]. A monadic style enables the programmer to cope with interaction and state-based computations in a functional setting. Also, higher-order structures can be defined, which mimic the control structures of imperative languages, and giving rise to the term *imperative functional programming* [6, 10]. However, due to the relationship between monads and category theory, and the proximity of the monadic style to the tempting realm of imperative programming, these advances have been mostly relegated to postgraduate courses.

We claim that it is completely viable to teach monadic algorithms to freshmen. Moreover, this can —and should— be done without explaining the technical details of monads. The benefits of accepting this challenge,

*Work partially supported by the spanish projects CAM-06T/033/96 and CICYT-TIC97-0672.

are two-fold: on the one hand, students are able to tackle a wider spectrum of programming applications; on the other hand, they learn imperative concepts without leaving the functional world.

The aim of the present paper is to substantiate our claim by explaining how to gently introduce the monadic style of programming to first-year students, and by providing some simple, yet illustrative, examples for this teaching task. We start with a brief presentation of the context where our proposal has sprouted. Then, we explain and detail a bit the proposal. Section 4 is the core of the paper, containing the teaching sequence we have followed and the corresponding monadic algorithms. We end by commenting some results from our experience.

2 The context

Before presenting our proposal, it is important to clearly explain the context and circumstances of our course. Attempts to give introductory programming courses based on functional programming languages have been sometimes forsaken for fear of a not completely satisfactory integration with the rest of the curriculum. Functional programming turns out to be so natural and close to problem-thinking, that students find difficulties to handle languages like FORTRAN or C when they are confronted to this low-level programming style in successor courses.

It is a reality that computer science curricula are mostly imperative programming oriented, with most subjects based on this style, while other programming paradigms are included as complementary or optional courses. For instance, while there is a great variety of first courses on programming from the functional perspective, there are very few proposals for a second course on programming (advanced data structures and program design methods) in a functional style (see [13] for a proposal).

Nonetheless, our proposal is not addressed to future computer engineers, but to first-year undergraduate mathematic students, which must follow a compulsory course on programming and, probably, will never learn anymore on computers or programming. Although, in our case, there is a second programming course on data structures and algorithms, this is only an option among a great and diversified offer on pure and applied mathematics subjects. Therefore, the main goal of this introductory course to programming is not to prepare students for later courses on the computing discipline, but to teach them how to use such a powerful and nowadays indispensable tool: a programming language. Of course, it is not our goal to teach a particular language and system, but to make the students to understand the main concepts in programming so that they will be able to design algorithms to solve their problems, and to express them in the available programming language —imperative in most cases. While the functional style is excellent for algorithm design, even more for mathematicians, the training would be incomplete without an understanding of the imperative computing model, and of the most typical data structures of the imperative style, i.e. arrays and files, which will be extensively used in subjects like numerical analysis or statistics.

A first attempt we tried to follow —inspired by the approach of [5]— was to present functional languages as a specification tool for describing algorithms, which could be directly executed, or which could be later efficiently implemented in an imperative language. Actually, Hartel and Muller, describe how to learn C after a first course on SML. A related experience is presented in [3], where Miranda is used for ADTs specifications to be implemented in C. In [7] a first-year course combining functional and imperative programming is described. Our project was not so ambitious, because we were constrained to a one-year course. Thus, 75% of the course was devoted to pure functional programming, while the remaining 25% was employed to explain, by using a conventional imperative language, the main imperative concepts (updatable variables, sequencing, iteration, arrays, files, subprograms). However, this first experience was quite a failure. The main reason was the scarce integration between the two programming styles. The methodology 'functional specification – imperative implementation' only worked for simple examples because many of the functional constructions, like non-tail recursion or higher-order functions, were difficult to translate in a systematic way

to the imperative style. We concluded that it was easier to design the algorithms directly using the imperative features. Consequently, the students 'divided' our course into two independent subjects: Haskell and Pascal, which were the languages chosen to be used in laboratories. This desintegration was aggravated by the lack of time: 60 classroom hours plus 30 hours in labs appeared to be too scanty to make them understand the two paradigms. Thus, while students were still fighting against higher-order functions, we suddenly started to talk about states and iterations. It is not the case that these concepts are difficult to grasp, but the students were unable to express them in the new syntax, and the confusion between the two notations was great.

3 The proposal

As we have explained in the previous section, the failure of our first experience was caused by the desintegration between the two programming styles, increased by the use of two different syntaxes. Hence, what about having the two programming models in a unique language? Then, we turned to the monadic programming style commented at the introduction of this paper.

Our actual proposal distributes the subject in a 75 % 'pure' functional + 25% 'imperative' functional. In this way, we still keep a quarter of the course devoted to the essentials of the imperative model:

- control of sequence;
- repetition, as an alternative to recursion; and
- a mutable state, allowing efficient data structures (arrays) and permanent data (files).

We have chosen Haskell as the supporting language because it includes all the features we desire to communicate to our students, while enjoying an easy to learn and handy syntax. Moreover, it is widely known in the functional language community, with much ongoing research on it, and providing very efficient compilers. There exist also the possibility of using an interpreter like HUGS, which allows the students to quickly test on the computer the examples learned at the classroom, and to easily develop small programs.

A detailed program is given next:

Part I: Introducing Programming
Lesson 1: Introduction. Algorithms and programs. Underlying hardware. Programming languages. Operating systems and translators.
Lesson 2: Program correctness. Program specification. Program design and verification.

Part II: Basic Functional Programming
Lesson 3: Basic types and simple expressions. Haskell: basic syntax and evaluation. Values and data types. Integers, floating point numbers, booleans, characters and strings.
Lesson 4: Function definitions. Conditional expressions and guards. Simple patterns. Function application. Function composition.
Lesson 5: Top-down design. Declaration scope. Programming with local definitions. Function refinement.
Lesson 6: Recursive functions. Mathematical induction. Recursive decomposition. Recursive functions over integers. Proof by induction.
Lesson 7: The type system. Introducing classes. A tour of the built-in Haskell classes. Monomorphic and polymorphic types. Type checking.
Lesson 8: Tuples. Concept. Value construction and patterns. Standard operations. Component selection and pattern matching.

Lesson 9: Lists. Concept. Value construction and patterns. Polymorphic lists. Standard operations. Recursive functions over lists. Proof by structural induction.
Lesson 10: Designing functions over lists. List traversals and searchs. Sorting lists: selection sort, insertion sort, merge sort. Analysis of correctness.
Lesson 11: Program efficiency. O-notation. Basic orders of efficiency. Time complexity analysis.
Lesson 12: Higher-order functions. Functions as arguments. Higher-order functions over lists: filtering, mapping and folding. Insertion sort revisited. Functions as values and results. Partial application. Sections and lambda abstractions. Currying and uncurrying.
Lesson 13: List comprehensions. Concept and syntax. Examples: primes, quicksort. List comprehension and higher-order functions.
Lesson 14: Introducing abstract data types. The ADT concept. Modules in Haskell. Examples: stacks, FIFO queues, and sets. Implementation using lists.

Part III: Imperative Functional Programming
Lesson 15: The imperative computing model. Updatable variables and states. Sequential composition and iteration. Relationship with the underlying hardware.
Lesson 16: Interactive input and output. Interactive keyboard input and screen output. Interactive programs with file input/output. Sequencing using `>>` and `>>=`. The `do` notation.
Lesson 17: Immutable arrays. Index types. The `Array` module. Array creation and subscripting. Useful functions over arrays. Examples: tabulating results, binary search, inserting in a sorted array, matrix product.
Lesson 18: Mutable arrays. The `ST` (Strict State Thread) module. Basic actions over `(ST s) a`. Constructing a mutable computation. Examples: insertion sort, bubble sort.

Notice that we introduce classes (Lesson 7). We find difficult for students to understand the type information provided by HUGS if they know nothing about Haskell classes. However, we restrict ourselves to explaining the most basic concepts, and we do not expect our students to create new classes. On the other hand, algebraic types are absent from the program presented here. The main use of algebraic types is the definition of recursive types (e.g. trees), which we think are better suited for a second year. The structures we expect our students to master are the linear ones: lists and arrays.

The last part of the course, the one devoted to imperative functional programming, starts (Lesson 15) with an introduction to typical imperative concepts, without mentioning the functional paradigm.

The expected advantages of this new approach reside not only in keeping the same syntax for the two styles, but also in keeping the same programming environment at the laboratories. This saves a lot of time and mistakes. Besides that, it allows the student to continue using, in the imperative part, the usual functional style for the non monadic functions, thus contributing to their maturity in the paradigm.

4 Imperative functional programming by example

This section contains a detailed presentation of the teaching sequence we have followed in the imperative part of the course, and a number of illustrative monadic and non monadic algorithms we have used to transmit the imperative concepts to the students.

4.1 Sequence and iteration: I/O interaction

The simplest imperative concept to start with is *sequential composition of actions*. For the first time in the course, we wonder about the specific order in which actions should be performed. Input/output interaction is an area in which the student can naturally appreciate that the control of this ordering is important.

Atomic actions We start by explaining output, the type `IO ()`, and the most elementary I/O action, the one doing nothing: `done::IO ()`. Then, we go on with other atomic output actions: writing a character, writing a string, writing a complete file, and so on. Then, we generalize to input, to the type `IO a` and its atomic actions: `return a`, reading a character, reading a line, reading a complete file, and so on. As in [16] and in [1, Chapter 10], we stress the difference between *defining* an I/O action and *performing* it.

Sequencing actions In order to be able to establish dialogues, some way of sequencing these elementary actions must be provided. First we introduce the sequential combinator `>>`:

```
main = putChar 'a' >> putChar 'b'
```

Once two actions have been combined, recursion provides the means to sequence a variable number of actions:

```
putStr ""     = done
putStr (c:cs) = putChar c >> putStr cs
```

When an I/O action returns a value different from `()`, some way must be provided so that the rest of the interaction can use this value. If we write

```
main = getChar >> putChar 'a'
```

the `>>` combinator simply ignores the value returned by the first action, so we justify the second combinator `>=`:

```
main = getChar >>= putChar
```

We explain the type `(>>=)::IO a -> (a -> IO b) -> IO b` and build more complex interactions:

```
getLine = getChar >>= \c -> if c=='\n' then return ""
                                       else getLine >>= \cs -> return (c:cs)
```

To explain these ideas we do not appeal to monads. For students, the `>>=` combinator is just read 'followed by'.

The `do`-notation At this point, the need for a more compact and clear notation is strongly felt, and we introduce the `do`-notation, explaining that this is just an abbreviation of the more cumbersome combination of `>>`, `>>=` and lambda abstractions:

```
getLine = do c <- getChar
             if c=='\n' then return ""
                        else do cs <- getLine
                                return (c:cs)
```

We could have chosen to explain only the `do`-notation, instead of presenting it as an abbreviation of more elementary concepts. But, in doing so, we could have transmitted the impression of a magical behaviour behind imperative-style algorithms. We have preferred to remark that programs are still functional.

Repetition Frequently in interactions, there is the need to repeat an action until some desired property holds. Here is an example of a program reading an integer between 1 and a given number `n`:

```
readInt :: Int -> IO Int
readInt n = do putStr ("Type an integer between 1 and " ++ show n ++ ": ")
               s <- getLine
               let x = read s in
                   if all isDigit s && 1 <= x && x <= n then return x
                                                        else readInt n
```

We tell the students that this construction is very typical in an imperative language and that there are special control structures such as `while` and `repeat` to express it.

Top down design of interactions Monadic dialogs should not look like long sequences of actions. Top down design has its place here. When a complex dialogue must be designed, it is advisable to split it into pieces, each one taking care of a part of the interaction. For instance, we can design a program performing the following loop: displaying a menu, inviting the user to choose an option, performing the corresponding action, and going back to the loop, or leaving it if the option chosen was the last one:

```
main = do showMenu
          i <- readInt 3
          case i of
            1 -> do {action1 ; main}
            2 -> do {action2 ; main}
            3 -> done
```

4.2 Read only state: immutable arrays

The next important imperative concept is the array data structure. It mimics the computer memory and so allows accessing to any single component in constant time, independently of the number of elements stored in the array. It is important that students understand the differences between this structure and lists: (i) once created, an array cannot be extended with new elements to produce a new array; and (ii) the recursion patterns for arrays are based on changing index intervals instead of on applying the recursive function to a substructure of the same type.

The type `Array a b` of immutable arrays is a good starting point for introducing later on mutable arrays. There are many algorithms, mainly reading from immutable arrays, having the same time complexity as if they were programmed in an imperative language. One of them is linear search:

```
linSearch :: Eq b => Array Int b -> b -> Maybe Int
linSearch a x = linSearch' a x low up
                where (low,up) = bounds a
linSearch' a x j up
  | j > up    = Nothing
  | x == a!j  = Just j
  | otherwise = linSearch' a x (j+1) up
```

We will always use the technique shown in this example, in every recursive definition related to either immutable or mutable arrays: the function to be designed is embedded in a more general one having at least two additional parameters, the following index to be dealt with, and the upper bound of this index. This more general function is recursively designed: a base case is reached when we get the empty interval of indices; in the recursive case, we decrease the length of the index interval. Of course, if the array is sorted, we can do it better by using a binary search:

```
binSearch :: Ord b => Array Int b -> b -> Maybe Int
binSearch a x = binSearch' a x low up
                where (low,up) = bounds a
binSearch' a x j k
  | j > k    = Nothing
  | x < a!m  = binSearch' a x j (m - 1)
  | x == a!m = Just m
  | x > a!m  = binSearch' a x (m + 1) k
  where m = (j + k) `div` 2
```

It is well known that this algorithm has logarithmic cost. We emphasize the fact by explaining that no search algorithm using lists as a search structure can beat this cost.

Other interesting algorithms with immutable arrays include matrix multiplication, Fibonacci tabulation, and the definition of higher order functions for arrays, similar to `map`, `fold`, `all`, `any` and so on. We also give a version of insertion sort for immutable arrays (whose cost is in $O(m^2)$, being m the length of the array):

```
isort :: Ord b => Array Int b -> Array Int b
isort a = foldl insert a [low+1..up]
          where (low,up) = bounds a
insert :: Ord b => Array Int b -> Int -> Array Int b
insert a n = insert' a low n
             where (low,_) = bounds a
insert' a j n
  | j >= n    = a
  | a!n > a!j = insert' a (j+1) n
  | otherwise = a // ((j,a!n) : [(k+1,a!k) | k <- [j..n-1]])
```

This algorithm will be the basis for a similar algorithm using mutable arrays. A call to `insert a n` assumes that the elements of `a` in positions `[low..n-1]` are ordered, and that `low` $<$ `n` $\leq$ `up`; then, it rearranges the elements in positions `[low..n]` in such a way that, at the end, this portion becomes ordered. In the worst case, each call to `insert` creates a new array by modifying the one given as parameter. This cost is in $O(m)$, being m the number of elements in the array. As there are $m-1$ calls to `insert`, the total cost of `isort` is in $O(m^2)$.

4.3 Read-write state: mutable arrays

Coming back to the analogy between arrays and the computer memory, it is easy to justify the need for mutable arrays: we would like to modify an array element, as we can do with a memory position, with a cost in $O(1)$. We explain that it is possible to express mutable arrays in a pure functional language such as Haskell provided two conditions are met:

- The programmer imposes a strict sequential order to the actions performed on a mutable array.
- The programmer accepts that, once a mutable array is modified, only the new copy is available to the remaining actions of the sequence. This implies to accept that a name *is connected to* different values in different parts of a text (we know that this fact does not violate transparential referency since a name denotes always the same mutable variable. What 'changes' is the state. More exactly, it is passed around from one action to the following one).

We explain that the tools for creating sequences of mutable actions are already known: the `>>` and `>>=` combinators, the `return` action, and the `do`-notation are not privative of the type `IO a`. We say that they are *overloaded* and that the type `ST s a` of mutable state actions can also enjoy of them (we say in passing that both constructors, `IO` and `ST s`, and some other, belong to the constructor class `Monad`).

In the following, we assume that the library module `ST`, standard in all Haskell distributions, which provides the interface to the mutable state actions proposed in Launchbury and Peyton Jones's paper [11] has been imported. We explain to the students the elementary mutable actions: creating a mutable array or a mutable variable, reading from them, writing to them, and so on. We also present the special function `runST::ST s a -> a` which is mandatory if we wish to encapsulate state-based computations into a non state-based one.

Our first algorithms use embedding and recursion on indices as we did with immutable arrays. Here is the mutable version of `insert`:

```
insert :: Ord b => STArray s Int b -> Int -> ST s ()
insert ma n = insert' ma low n
              where (low,_) = boundsSTArray ma
insert' :: Ord b => STArray s Int b -> Int -> Int -> ST s ()
insert' ma j n
  | j >= n    = return ()
  | otherwise = do a_n <- readSTArray ma n
                   a_j <- readSTArray ma j
                   if a_n > a_j then insert' ma (j+1) n
                                else do shift ma j (n-1)
                                        writeSTArray ma j a_n
```

The reader is invited to compare this program with the one given in Section 4.2. The similarities are obvious. The big difference is that now, as we are working with only one array instead of with two, the order in which modifications to the array are performed is crucial. Once we have found that element `a_n` must go into position `j`, we must first shift the elements between position `j` and position `n-1` one place to the right and then write `a_n` into position `j`. Should we change this order, the array would become corrupted. The shifting action can also be defined by recursion on indices. We will present a higher order version of `shift` in Section 4.4. The cost of `insert` is clearly in $O(m)$, being m `= n - low + 1` the length of the array portion affected by insertion. Every position in this portion is subject to a read or/and a write operation, each one with a cost in $O(1)$.

For the complete insertion sort algorithm, we cannot use `foldl` because the types do not match. We cannot either use the monadic version of `foldl`, called `foldM::Monad m => (a -> b -> m a) -> a -> [b] -> m a`, for much the same reason. For the moment, we content ourselves with a recursive version:

```
mutIsort :: Ord b => STArray s Int b -> ST s ()
mutIsort ma = mutIsort' (low+1) up ma
              where (low,up) = boundsSTArray ma
mutIsort' :: Ord b => Int -> Int -> STArray s Int b -> ST s ()
mutIsort' j up ma
  | j > up    = return ()
  | otherwise = do insert ma j
                   mutIsort' (j+1) up ma
```

If the programmer wishes to hide the whole stateful computation, he can use `runST` to encapsulate it:

```
isort :: Ord b => Array Int b -> Array Int b
isort a = runST (do ma <- thawSTArray a
                    mutIsort ma
                    a' <- unsafeFreezeSTArray ma
                    return a')
```

The function first converts an immutable array into a mutable one, sorts it, and saves its final state into a new immutable array which is returned as result. From the outside, the algorithm looks like sorting immutable arrays.

4.4 Higher order abstractions

Functional programming is known to be good for abstracting common computation patterns into higher order functions. In the area of monadic algorithms, useful computation patterns more or less correspond to control structures present in most imperative languages.

Simulating an imperative for-loop The first useful abstraction is the predefined function

```
sequence :: Monad m => [m a] -> m ()
sequence = foldr (>>) (return ())
```

converting a list of monadic actions into a single action which performs sequentially the actions in the list. Used in combination with `map`, it can serve as a good simulation of the **for** control structure of many imperative languages. Consider the expresion `sequence (map f indices)`. Function `map` creates a list of actions by mapping a function, depending on an index, to the list of indices; `sequence` threads the action list into a single action. So, by providing and appropriate list of indices and a 'body' function we get a functional equivalent of the imperative **for**. Here is the higher order implementation of function `shift` in Section 4.3:

```
shift :: STArray s Int b -> Int -> Int -> ST s ()
shift ma i j = sequence (map action [j,j-1..i])
               where action k = do x <- readSTArray ma k
                                   writeSTArray ma (k+1) x
```

Notice the order in which positions are shifted. Likewise, here is the higher order version of `mutIsort` of Section 4.3:

```
mutIsort :: Ord b => STArray s Int b -> ST s ()
mutIsort ma = sequence (map (insert ma) [low+1..up])
              where (low,up) = boundsSTArray ma
```

If the teacher wishes to use a style with a more imperative flavour, he can define

```
for :: Monad m => [a] -> (a -> m ()) -> m ()
for indices body = sequence (map body indices)
```

and translate the above examples to use this construction. A slightly different `for` function was originally proposed in [12]. For instance, the `shift` function would look like:

```
shift ma i j = for [j,j-1..i] action
               where action k = ...
```

But we claim that for functional programmers (e.g. our students) the direct use of `sequence` and `map` is more illustrative than that of `for`.

General linear search When working with mutable arrays, useful abstractions include the corresponding versions of `map`, `fold`, `any`, `all`, and so on. Another interesting abstraction is looking for the first array element satisfying a given property, i.e. a generalization of linear search:

```
gLinSearch :: STArray s Int b -> (b -> Bool) -> ST s (Maybe Int)
gLinSearch ma p = gLinSearch' ma p low up
                  where (low,up) = boundsSTArray ma
gLinSearch' ma p j up
  | j > up    = return Nothing
  | otherwise = do a_j <- readSTArray ma j
                   if p a_j then return (Just j)
                            else gLinSearch' ma p (j+1) up
```

By using it, we can write a very compact version of the mutable `insert` function of Section 4.3:

```
insert ma n = do a_n <- readSTArray ma n
                 ~(Just j) <- gLinSearch ma (a_n <=)
                 shift ma j (n-1)
                 writeSTArray ma j a_n
```

Notice that, in the worst case, the search ends up with `j = n`. In this case, the shift action just does nothing, and writing `a_n` into position `n` produces no harm. The irrefutable pattern in the second line is a requirement of the `do`-notation.

Simulating an imperative while-loop The last abstraction we present is a kind of **while** loop. Differently from the one presented in [14, Chapter 14] for the type `IO`, we have found that the action in the body is usually different from one iteration to the next, so we propose to give a list of actions as the second argument:

```
while :: Monad m => m Bool -> [m ()] -> m ()
while test []     = return ()
while test (a:as) = do continue <- test
                       if continue then do {a ; while test as}
                                   else return ()
```

The loop ends either when the test fails or when the list of actions is —if ever— exhausted. By using it, we can write a higher order version of the well known bubble sort algorithm:

```
bubbleSort :: Ord b => STArray s Int b -> ST s ()
bubbleSort ma = do boolVar <- newSTRef True
                   while (readSTRef boolVar) (map (stage boolVar up)
                                                  [low..up-1])
  where (low,up) = boundsSTArray ma
        stage v up k = do writeSTRef v False
                          sequence (map (action v) [up-1,up-2..k])
        action v j   = do x <- readSTArray ma j
                          y <- readSTArray ma (j+1)
                          if x <= y then return ()
                                    else do -- array is being changed
                                            writeSTArray ma j     y
                                            writeSTArray ma (j+1) x
                                            writeSTRef v True
```

For an array with n elements, the algorithm performs, in the worst case, $n-1$ stages, with index `k` ranging from `low` to `up-1`. At the end of stage `k` we have at position `k` the next minimum element of the array. So, the array gets sorted from left to right. We make use of a mutable boolean variable `boolVar` to record whether there has been any modification to the array in the current stage. If not, the test fails in the next iteration, the `while` loop is exited, and the whole computation terminates. This means that, for an initially sorted array, bubble sort performs an only stage, with a time complexity in $O(n)$.

4.5 Putting all together

At the end of the course, students should be able to combine imperative functional programming with classical functional programming. So, in order to know if they have acquired these skills, we have proposed them to write, as a final laboratory assignment, a program whose core is the Floyd algorithm [4]. The aim of this algorithm is to compute the shortest paths between each pair of nodes of a given graph. It receives the graph as input, and generates as output two bidimensional arrays: one to record the shortest distance between each pair of nodes; and the other to store the necessary information to obtain the shortest paths. This is a dynamic programming problem and, of course, first-year students are not expected to discover it by themselves. Instead, we explain to them in words what has to be done to solve the problem, and then they have to implement it.

We have chosen this example because it combines all the features we have taught in the course:

- There are several *I/O operations*, and it is important to perform them in the right sequence: at the beginning, the original graph is to be read from a file; after computing the arrays, the program interacts with the user, who can ask for the shortest path between any pair of nodes.
- It is convenient to use *mutable arrays*, because the core of the algorithm is a loop that computes the paths by refining the solutions found so far. At each stage k, for each pair of nodes (i,j) it is decided if a better path between i and j can be obtained by visiting node k as an intermediate step. Each time a better path is found, both arrays are modified.
- After computing the arrays, there is no need to modify them anymore. Therefore, *immutable arrays* can be used.
- It is easier and clearer to express Floyd's algorithm by using *higher order functions* than by using recursion.

Assuming that the original graph is represented by a matrix in which position (i,j) contains ∞ if the nodes are not directly connected, and contains the distance of the connection otherwise, a compact and precise way to write the algorithm is:

```
-- Encapsulates the mutable computations of the program
floydAlg :: Array (Int,Int) Int -> (Array (Int,Int) Int, Array (Int,Int) Int)
floydAlg t = runST (do tm <- thawSTArray t
                       um <- newSTArray (bounds t) 0
                       floyd tm um
                       ti <- unsafeFreezeSTArray tm
                       ui <- unsafeFreezeSTArray um
                       return (ti,ui))

-- Floyd algorithm using two mutable arrays
floyd :: STArray a (Int,Int) Int -> STArray a (Int,Int) Int -> ST a ()
floyd tm um = sequence (map stage [l..u])
  where ((l,l'),(u,u')) = boundsSTArray tm
        stage k         = sequence (map (refine k) (range ((l,l'),(u,u'))))
        refine k (i,j)  = do tij <- readSTArray tm (i,j)
                             tik <- readSTArray tm (i,k)
                             tkj <- readSTArray tm (k,j)
                             if tik + tkj < tij
                                then do writeSTArray tm (i,j) (tik + tkj)
                                        writeSTArray um (i,j) k
                                else return ()
```

We have found out that our students are able to write programs similar to the solution given above, and that they understand the conceptual differences between 'normal' operations and operations involving a state.

5 Results

We only report here the results relevant to the subject of this paper. General results about the use of a functional language in a first-year course have been reported elsewhere (see, for example, [2]).

At the time of writing these lines we can assess whether part of the goals of the course has been met or not but, unfortunately, we cannot do it for all of them. In particular, it is very early to know which kind of difficulties these student will have when confronted, in the next few years, to actual imperative languages

such as C or Pascal. Will the concepts learned in our course be enough to understand the new languages? Will they recognize the imperative model of computation in spite of the change of syntax? Will they easily replace recursion by iteration? We plan to follow the evolution of these students in the next two years to collect information about this aspect but, for now, we can only guess what may happen.

For the moment, through their laboratory assignments and written examinations, we have collected enough information to assess the quality of the skills they have acquired. The most important conclusion relevant to this paper is that we have *not* detected the students to have special difficulties with monadic algorithms. In particular, they accept very naturally the concepts of sequential actions and of mutable state.

With respect to sequential composition, we think that the `do`-notation, proposed originally in [10], deserves most of the merit for it. It is very simple, illustrative of what is going on, and hides a lot of clumsy details that the students are happy to ignore. In our opinion, it has been a very good decision to include it as part of Haskell.

However that, and perhaps because the `do`-notation is a high level abstraction, the students tend to confuse the `<-` in a `do` sequence with the `=` in an equation, and produce programs in which they mix both notations, such as the following one:

```
main = do x <- action
          y = f x
          ...
```

The confusion is favoured by the fact that the syntax `<-` is also used in list comprehensions, with a second meaning. The essence of the problems is that they do not see a clear difference between the type `IO a` and the type `a`. This question —Why are they different?— has been very frequently asked to us. Fortunately, the type system takes care of these mistakes and forces them to use the correct syntax. The question has also to be with understanding the `>>=` combinator underlying the syntax `x <- action`. We have found that this combinator is much more difficult to understand than the `>>` one. For this reason, we think that perhaps it is a good approach to move quickly from using raw `>>=` and lambda abstractions to the `do`-notation.

Another interesting result is that higher order abstractions, such as those proposed in Section 4.4, are very easily apprehended in this part of the course. For instance, they are willing to give up recursion on behalf of using the `sequence-map` combination, when they detect that the same action has to be repeated for a set of indices. This is in contrast to what has happened in the rest of the course, where they are strongly reluctant to use higher order functions (in particular, those of the `fold` family).

In summary, we think that the approach followed here can be useful for those having context conditions similar to ours: (i) you believe that functional programming has didactic advantages over imperative programming for first-year students; (ii) your students need also to understand the imperative model of computation to be able to learn imperative languages in subsequent courses; (iii) there is no much time available for the course.

References

[1] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2 edition, 1998.

[2] C. Clack and C. Myers. The dys-functional student. In *LNCS 1022*, pages 289–309. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[3] A. Davison. Teaching C after Miranda. In *LNCS 1022*, pages 35–50. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[4] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[5] P. Hartel and H. Muller. *Functional C*. Addison-Wesley, 1997.

[6] S. Peyton Jones and P. Wadler. Imperative functional programming. In *ACM Principles of Programming Languages*. ACM, 1993. Charleston, N. Carolina.

[7] S. Joosten, K. van den Berg, and G. van der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3:49–65, 1993.

[8] E. T. Keravnou. Introducing computer science undergraduates to principles of programming through a functional language. In *LNCS 1022*, pages 15–34. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[9] T. Lambert, P. Lindsay, and K. Robinson. Using Miranda as a first programming language. *Journal of Functional Programming*, 3:5–34, 1993.

[10] J. Launchbury. Lazy imperative programming. In *ACM Workshop on State in Programming Languages*, pages 1–11, 1993.

[11] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation, PLDI'94*, pages 24–35, June 1994.

[12] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995. Elaboration of [11].

[13] M. Núñez, P. Palao, and R. Peña. A Second Year Course on Data Structures based on Functional Programming. In *LNCS 1022*, pages 65–84. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[14] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[15] P. Wadler. The essence of functional programming. In *19'th Symposium on Principles of Programming Languages*. ACM, January 1992. Alburquerque, New Mexico.

[16] P. Wadler. How to declare an imperative. In *International Logic Programming Symposium*. MIT Press, 1995.

[17] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming. LNCS 925*. Springer-Verlag, 1995.

Modular Lazy Search for Constraint Satisfaction Problems*

Thomas Nordin Andrew Tolmach
Pacific Software Research Center
Oregon Graduate Institute & Portland State University
nordin@cse.ogi.edu apt@cs.pdx.edu

Abstract

We describe a unified, lazy, declarative framework for solving constraint satisfaction problems, an important subclass of combinatorial search problems. These problems are both practically significant and hard. Finding good solutions involves combining good general-purpose search algorithms with problem-specific heuristics. Conventional imperative algorithms are usually implemented and presented monolithically, which makes them hard to understand and reuse, even though new algorithms often are combinations of simpler ones. Lazy functional languages, such as Haskell, encourage modular structuring of search algorithms by separating the generation and testing of potential solutions into distinct functions communicating through an explicit, lazy intermediate data structure. But only relatively simple search algorithms have been treated in this way in the past.

Our framework uses a generic generation and pruning algorithm parameterized by a labeling function that annotates search trees with conflict sets. We show that many advanced imperative search algorithms, including backmarking, conflict-directed backjumping, and minimal forward checking, can be obtained by suitable instantiation of the labelling function. More importantly, arbitrary combinations of these algorithms can be built by simply composing their labelling functions. Our modular algorithms are as efficient as the monolithic imperative algorithms in the sense that they make the same number of consistency checks, and most of our algorithms are within a constant factor of their imperative counterparts in runtime and space usage. We believe our framework is especially well-suited for experimenting to find good combinations of algorithms for specific problems.

1 Introduction

Combinatorial search problems offer a great challenge to the academic researcher: they are of tremendous interest to commercial users, and they are often very computationally intensive to solve. Over the past several decades the AI community has responded to this challenge by producing a steady stream of improvements to generic search algorithms. There have also been numerous attempts to organize the various algorithms into standardized frameworks for comparison (e.g., [8, 6, 16]).

While the speed and cunning of the search algorithms have improved, the new algorithms are more complicated and harder to understand, even though they are often combinations of simpler standard algorithms. The problem is exacerbated by the fact that most algorithms are described by large, monolithic chunks of pseudo-code (or C code). Although it is recognized that most problems benefit from a tailor-made solution, involving a combination of existing generic and domain-specific algorithms, modularity has not been a strong point of much of the recent research. It is difficult to reuse code except via cut-and-paste. Moreover, to prove these algorithms correct we must resort to complex reasoning about their dynamic behavior.

*Work supported, in part, by the US Air Force Materiel Command under contract F19628-26-C-0161.

For example, although most of these search algorithms are conceived as varieties of "tree search," no actual tree data structures appear in their implementations; only virtual trees are present, in the form of recursive routine activation histories. Perhaps for this reason, even widely-used and well-studied algorithms often lack correctness proofs.

In the lazy functional programming world, the idea of implementing a search algorithm using modular techniques is a commonplace. The classic paper of Hughes [9] and text of Bird and Wadler [3] both give examples of search algorithms in which generation and testing of candidate solutions are separated into distinct phases, glued together using an explicit, lazy, intermediate data structure. This "generate-and-test" paradigm makes essential use of laziness to synchronize the two functions (really coroutines) in such a way that we never need to store much of the (exponential-sized) intermediate data structure at any one time. In general, the modular lazy approach can lead to algorithms that are much simpler to read, write, and modify than their imperative counterparts. However, the algorithms described in these sources are fairly elementary.

In this paper we present a lazy declarative framework for solving one important class of combinatorial search problems, namely constraint satisfaction problems (CSPs). For simplicity, we restrict our attention to binary constraint problems, and to search algorithms that use a fixed variable order; neither of these restrictions is fundamental. Our framework is based on explicit, lazy, tree structures, in which each tree node represents a *state* in the search space; a subset of the tree's leaf nodes corresponds to problem solutions. Nodes can be labeled with *conflict sets*, which record constraint violations in the corresponding states; many algorithms use these sets to *prune* subtrees that cannot contribute a solution.

Our framework is written in Haskell. We provide a small library of separate functions for generating, labeling, rearranging, pruning, and collecting solutions from trees. In particular, we describe a generic search algorithm, parameterized by a labeling function, and show that a variety of standard imperative CSP algorithms, including simple backtracking, backmarking, conflict-directed backtracking, and forward checking, can be obtained by making a suitable choice of labeling function. Using an explicit representation of the search tree allows us to think about the intermediate values and gives us new insights into more efficient algorithms. As in recent work on functional data structures[10, 14], we found that recasting imperative algorithms into a declarative lazy setting casts new light on the fundamental algorithmic ideas. In particular, it is easy to see how to combine our algorithms, simply by composing their labeling functions, and to see that the result will be correct.

Since the whole point of improving search algorithms is to be able to solve larger problems faster, we must obviously be concerned with the performance of our lazy algorithms. Our experiments show that lazy, modular Haskell code is several times slower than strict, manually integrated Haskell code; moreover, even the latter can be an order of magnitude slower than highly optimized C code. However, since search times often explode exponentially, even slowdowns of one or two orders of magnitude have little effect on the size of problem we can solve within a fixed time bound. All our algorithms and their combinations are fast enough for experiments that have been interesting to researchers in the past; for example we are able to reproduce parts of the tables in [2, 11]. More importantly, our implementations are fast enough to allow experimentation with different combinations of algorithms on problems of realistic size. For such experiments, CPU time is often not an ideal comparison metric, since it is difficult to compare numbers obtained from different implementations on different systems. A widely used alternative metric is the number of consistency checks performed by the algorithm.

The remainder of the paper is organized as follows. Section 2 describes our problem domain and Section 3 gives a Haskell specification for it. Section 4 describes simple tree-based backtracking search. Section 5 introduces conflict sets and our generic search algorithm, and recasts backtracking search in that framework. Section 6 briefly discusses search heuristics. Sections 7, 8, and 9 describe more sophisticated algorithms, and Section 10 discusses their combination. Section 11 summarizes performance results, Section 12 describes related work, and Section 13 concludes.

The reader is assumed to have a working knowledge of functional programming, and some familiarity

with laziness. Peculiarities of Haskell syntax will be explained as they arise. All the code examples in this paper are available on the World Wide Web at `http://www.cs.pdx.edu/~apt/CSP.hs`.

2 Binary Constraint Satisfaction Problems

A *binary constraint satisfaction problem* is:

- a set of variables $V = \{v_1, v_2, \ldots v_n\}$;
- for each variable v_i, a finite set D_i of possible values (its *domain*);
- and a set C of pairwise *constraints* between variables that restrict the values that they can take on simultaneously.

Each constraint is a relation on two named variables, i.e., a triple (i, j, R) where $R \subseteq D_i \times D_j$.

An *assignment* v_i`:=`x_i associates a variable v_i to some value $x_i \in D_i$. A *state* is a collection of assignments for a subset of V. A state $\{\ldots v_i$`:=`$x_i, \ldots v_j$`:=`$x_j, \ldots\}$ *satisfies* a constraint (i, j, R) if $(x_i, x_j) \in R$. A state is *consistent* if it satisfies every constraint on its variables, i.e., if for every pair of assignments v_j`:=`x_j, v_k`:=`x_k in the state, and every matching constraint (j, k, R) in C, $(x_j, x_k) \in R$. A state is *complete* if it assigns all the variables of V; otherwise it is *partial*. A *solution* to a CSP is any complete consistent state. For some problems we want to calculate all solutions, but for many we only wish to find the first solution as quickly as possible.

In this paper, we fix the variable order $v_1, v_2, \ldots v_n$, i.e., we consider only states such that if v_i is in the state so is v_j for all $j < i$. We define the *level* of a variable v_i to be i and the level of a state to be the maximum of its variables' levels. To simplify the presentation, we further assume that all domains have the same size m and that their values are represented by integers in the set $\{1, 2, \ldots, m\}$.

A naive approach to solving a CSP is to enumerate all possible complete states and then check each in turn for consistency. In a binary CSP, consistency of a state can be determined by performing *consistency checks* on each pair of assignments in the state, until an *inconsistent pair* of variables is detected, or all pairs have been checked. Following the conventions of the search literature, we use the number of consistency checks as a key measure of execution code, although it is not necessarily an accurate measure unless each check can be performed in unit time, which is not the case for all problems.

3 CSPs in Haskell

Figure 1 gives a Haskell framework for describing CSP problems and an implementation of a naive solver. An assignment is constructed using the infix constructor `:=`. A CSP is modeled as a record containing the number of variables, `vars`, the size of their domain, `vals`, and a constraint oracle, `rel`. We represent the oracle as a Haskell function taking two assignments and returning `False` iff the assignments violate some constraint. This function can be implemented by a four-dimensional array of booleans or by a mathematical formula.

We present the solver in the standard "lazy pipeline" style that separates generation of candidate solutions (here the set of all complete states) from consistency testing. States are represented as lists of assignments sorted in decreasing order by variable number. Although this code appears to produce a huge intermediate list data structure `candidates`, lazy evaluation insures that list elements are generated only on demand, and elements that fail the filter in `test` can be immediately garbage collected. Similarly, although `inconsistencies` appears to build a list of *all* inconsistent variable pairs in the state[1], `consistent`

[1]This function uses a Haskell *list comprehension*, which is similar to a familiar set comprehension: this one builds a list of pairs of variable levels such that the corresponding assignments are drawn from the current state and are in conflict according to `rel`.

```
type Var = Int
type Value = Int

data Assign = Var := Value deriving (Eq, Ord, Show)

type Relation = Assign -> Assign -> Bool

data CSP = CSP { vars, vals :: Int, rel :: Relation }

type State = [Assign]

level :: Assign -> Var
level (var := val) = var

value :: Assign -> Value
value (var := val) = val

maxLevel :: State -> Var
maxLevel [] = 0
maxLevel ((var := val):_) = var

complete :: CSP -> State -> Bool
complete CSP{vars} s = maxLevel s == vars

generate :: CSP -> [State]
generate CSP{vals,vars} = g vars
  where g 0 = [[]]
        g var = [ (var := val):st | val <- [1..vals], st <- g (var-1) ]

inconsistencies :: CSP -> State -> [(Var,Var)]
inconsistencies CSP{rel} as =
  [ (level a, level b) | a <- as, b <- reverse as, a > b, not (rel a b) ]

consistent :: CSP -> State -> Bool
consistent csp = null . (inconsistencies csp)

test :: CSP -> [State] -> [State]
test csp = filter (consistent csp)

solver :: CSP -> [State]
solver csp  = test csp candidates
  where candidates = generate csp

queens :: Int -> CSP
queens n = CSP {vars = n, vals = n, rel = safe}
  where safe (i := m) (j := n) = (m /= n) && abs (i - j) /= abs (m - n)
```

Figure 1: A formulation of CSPs in Haskell.

```
data Tree a = Node a [Tree a]

label :: Tree a -> a
label (Node lab _) = lab

type Transform a b = Tree a -> Tree b

mapTree  :: (a -> b) -> Transform a b
mapTree f (Node a cs) = Node (f a) (map (mapTree f) cs)

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f (Node a cs) = f a (map (foldTree f) cs)

filterTree :: (a -> Bool) -> Transform a a
filterTree p = foldTree f
  where f a cs = Node a (filter (p . label) cs)

prune :: (a -> Bool) -> Transform a a
prune p = filterTree (not . p)

leaves :: Tree a -> [a]
leaves (Node leaf []) = [leaf]
leaves (Node _ cs) = concat (map leaves cs)

initTree :: (a -> [a]) -> a -> Tree a
initTree f a = Node a (map (initTree f) (f a))
```

Figure 2: Trees in Haskell.

actually demands only the head of the list (to check whether the list is `null`). Thus the solver actually calculates only the *earliest inconsistent pair* of variables for each state. Finally, although the solver returns a list of all solutions if demanded, it can be used to obtain just the first solution (and do no further computation) by asking for just the head of the result. Although the code thus uses much less space than a strict reading would suggest, this solver is still extremely inefficient because it duplicates work, but it is useful to illustrate lazy coding style and as a specification for the more sophisticated solvers we introduce below.

A simple problem useful for illustrating different search strategies is the n-queens problem, that is, trying to put n queens on a $n \times n$ chess board such that no queen is threatening another. using the standard optimization that we only try to place one queen in each column [13]. Given the definition of `queens`, we can apply the general-purpose CSP machinery to solve it; for example, the expression `solver (queens 5)` generates a list of solutions to the 5-queens problem.

4 Backtracking and Tree Search

The most obvious defect of the naive solver is that it can duplicate a tremendous amount of work by repeatedly checking the consistency of assignments that are common to many complete states. We say state S' *extends* state S if it contains all the assignments of S together with zero or more additional assignments. A fundamental fact about CSP's is that no extension to an inconsistent state can ever be consistent, so there is no point in searching such an extension for a solution. This observation immediately suggests a better solver algorithm. A *backtracking* solver searches for solutions by constructing and checking *partial* states, beginning with the empty state and extending with one assignment at a time. Whenever the solver discovers an inconsistent state, it immediately *backtracks* to try a different assignment, thus avoiding the fruitless exploration of that state's extensions. Moreover, consistency of each new state can be tested just by comparing

```
mkTree :: CSP -> Tree State
mkTree CSP{vars,vals} = initTree next []
  where next ss = [ ((maxLevel ss + 1) := j):ss | maxLevel ss < vars, j <- [1..vals] ]

data Maybe a = Just a | Nothing deriving Eq

earliestInconsistency :: CSP -> State -> Maybe (Var,Var)
earliestInconsistency CSP{rel} [] = Nothing
earliestInconsistency CSP{rel} (a:as) =
  case filter (not . rel a) (reverse as) of
    [] -> Nothing
    (b:_) -> Just (level a, level b)

labelInconsistencies :: CSP -> Transform State (State,Maybe (Var,Var))
labelInconsistencies csp = mapTree f
    where f s = (s,earliestInconsistency csp s)

btsolver0 :: CSP -> [State]
btsolver0 csp =
  (filter (complete csp) . leaves . (mapTree fst) . prune ((/= Nothing) . snd)
                                          . (labelInconsistencies csp) .  mkTree) csp
```

Figure 3: Simple backtracking solver for CSPs.

the newly added assignment to all previous assignments in the state, since any inconsistency involving *only* the previous assignments would already have been discovered earlier. If the solver manages to reach a complete state without encountering an inconsistency, it records a solution; if multiple solutions are wanted, it backtracks to find the others.

Backtracking solvers can be viewed very naturally as searching a *tree*, in which each node corresponds to a state and the descendents of a node correspond to extensions of its state. In conventional imperative implementations of backtracking, the tree is not explicit in the program; if a recursive implementation is used, the tree is isomorphic to the dynamic activation history tree of the program, but usually the tree is little more than a metaphor for helping the programmer reason informally about the algorithm. In the lazy functional paradigm it is natural to treat search trees as *explicit* data structures, i.e., programs are constructed as pipelines of operations that build, search, label, manipulate, and prune actual trees. As before, we rely on laziness to avoid actually building the entire tree.

Figure 2 gives Haskell definitions for a tree datatype and associated utility functions. A `Tree` is a node containing a label and a list of children, themselves `Tree`s. `mapTree`, `foldTree`, and `filterTree` are the analogues of the familiar functions on lists. `leaves` extracts the labels of the leaves of a tree into a list in left-to-right order. `initTree` generates a tree from a function that computes the children of a node [9].

The code in Figure 3 uses these trees to implement a backtracking solver `btsolver0` using a lazy pipeline. All the algorithms discussed in this paper expect the tree to be generated and maintained in fixed variable order, so that nodes at level i of the tree (counting the root as level 0) always extend their parent by an assignment to v_i. Thus, the generator, `mkTree`, works by providing a `next` function to `initTree` that generates one extension for each possible value of the next variable. Each node describes an entire (partial) state, but (in any reasonable Haskell implementation) it actually stores only a single assignment, together with a pointer to the remainder of the state embedded in its parent node.

The application (`labelInconsistencies` *csp*) returns a *tree transformer*: it adds an annotation to each node recording its earliest inconsistent pair (if any), as returned by `earliestInconsistency`. The standard tree function `prune` p removes nodes for which predicate p is true; in this instance it prunes all inconsistent nodes. The annotations are then removed by (`mapTree fst`). Any nodes representing

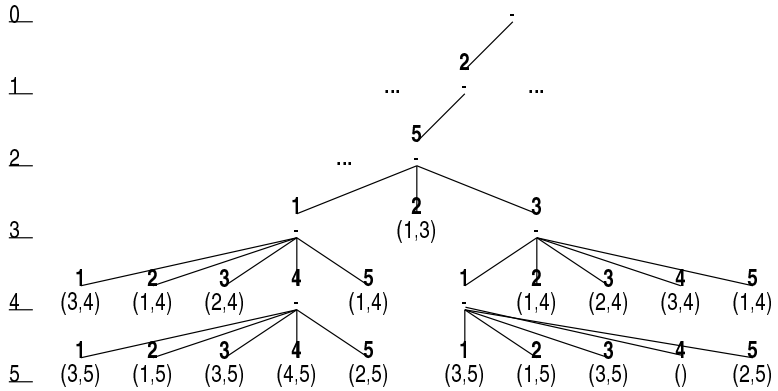


Figure 4: Portion of search tree for `queens 5`. Nodes at level i are annotated with their assigned value x_i (in **bold**), and with their earliest inconsistent pair, if any.

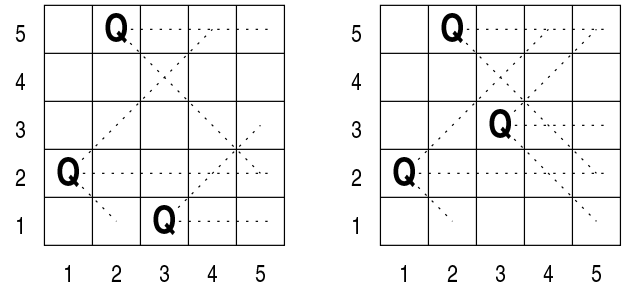


Figure 5: Two positions from the `queens 5` search tree in Figure 4. The left and right diagrams correspond to the left-most and right-most subtrees of level 3, respectively.

complete states that are still left in the tree must be solutions; the remaining pipeline stages extract these using the standard tree function `leaves` and the standard list function `filter`. Figure 4 illustrates the labels produced by `btsolver0` on part of the tree for `queens 5`; the corresponding board positions are shown in Figure 5. Note that the children of inconsistent nodes have been pruned.

It is essential to note that this pipeline is demand driven: each stage executes only when demanded by the following stage. In particular, inconsistency calculations will *not* be performed on nodes of the tree excised by `prune`, because the values of these nodes will never be demanded. Thus we get the desired effect of backtracking without any explicit manipulation of control flow. Also, as before, only a small part of each intermediate tree is ever "live" (non-garbage data) at any one time, namely the spine of the tree from root to current node, i.e., essentially what would be stored in activation records for a recursive imperative implementation. So our lazy algorithms pay at worst a constant factor more space than their imperative counterparts. We do, however, pay some overhead for building, storing, and garbage collecting each tree node, and, unless our Haskell implementation performs effective deforestation [7], this cost will be repeated for each intermediate tree in the pipeline. For these reasons, the lazy implementation of backtracking is about four times slower than a monolithic, strict Haskell implementation (see Section 11).

```
data ConflictSet = Known [Var] | Unknown deriving Eq

knownConflict :: ConflictSet -> Bool
knownConflict (Known (a:as)) = True
knownConflict _              = False

knownSolution :: ConflictSet -> Bool
knownSolution (Known []) = True
knownSolution _          = False

checkComplete :: CSP -> State -> ConflictSet
checkComplete csp s = if complete csp s then Known [] else Unknown

type Labeler = CSP -> Transform State (State, ConflictSet)

search :: Labeler -> CSP -> [State]
search labeler csp =
  (map fst . filter (knownSolution . snd) . leaves .
                            prune (knownConflict . snd) . labeler csp . mkTree) csp

bt :: Labeler
bt csp = mapTree f
  where f s = (s,
               case earliestInconsistency csp s of
                 Nothing    -> checkComplete csp s
                 Just (a,b) -> Known [a,b])

btsolver :: CSP -> [State]
btsolver = search bt
```

Figure 6: Conflict-directed solving of CSPs.

5 Conflict Sets and Generic Search

The utility of the backtracking solver is based on its ability to prune subtrees rooted at inconsistent nodes; it does nothing with consistent nodes. Of course, just because a state is consistent doesn't mean it can be extended to a solution; the assignments already made may be inconsistent with any possible choices for future variables. Figure 4 shows an example for `queens 5`: the assignment to value 1 at level 3 of the left-hand tree is consistent, but cannot be extended to a solution.

If a solver could identify such *conflicted* states, it could prune their subtrees too. To make precise the exact conditions under which such pruning is possible, we use the following definition. A *conflict set* for a state is a subset of (the indices of) the variables assigned by the state such that *any* solution must assign a *different* value to at least one member of the subset. More formally, given a state $S = \{v_1 := x_1, v_2 := x_2, \ldots, v_k := x_k\}$, a conflict set CS for S is a subset of $\{1, 2, \ldots, k\}$ such that, if $\{v_1 := y_1, v_2 := y_2, \ldots, v_n := y_n\}$ is a solution, then $(\exists i \in CS) x_i \neq y_i$. (Thinking imperatively, we might say a conflict set contains variables at least one of which "must be changed" to reach a solution.) Note that conflict sets are not, in general, uniquely defined. In particular, if a state at level k has a non-empty conflict set CS, then every subset of $\{1, \ldots, k\}$ containing CS is also a conflict set. If a state has a non-empty conflict set then no extension of that state can be a solution; conversely, if it has an empty conflict set, then it must have at least one extension that is a solution. This is a very strong characterization of states: for example, if we could compute a conflict set for the root of the tree (the empty state), we could test whether it were empty and thereby determine whether the problem has a solution at all! We will therefore often operate in an environment where many conflict sets are unknown. It is obviously not possible to identify a conflicted, but consistent, state without exploring *some* of its extensions;

```
hrandom :: Int -> Transform a a
hrandom seed (Node a cs) = Node a (randomList seed' (zipWith hrandom (randoms seed') cs))
  where seed' = random seed

btr :: Int -> Labeler
btr seed csp = bt csp . hrandom seed
```

Figure 7: A randomization heuristic

the trick is to avoid exploring all of them, and save effort by pruning the remainder. We address algorithms with this property beginning in Section 7.

For the moment, note that any inconsistent state has a non-empty conflict set. In particular, if a state has an earliest inconsistent pair (i, j) then it has $\{i, j\}$ as a conflict set, which we call the *earliest conflict set*. So we can subsume backtracking search in a more general algorithm we call conflict-directed search, shown in Figure 6. We define a generic routine `search`, parameterized by a `labeler` function, which annotates nodes with conflict sets. More precisely, if the labeler can determine a legal conflict set s for the node, it annotates the node with `Known` s; otherwise, it annotates it with `Unknown`. (In general, we also permit the labeler to rearrange or prune its input tree, so long as its output tree is properly labeled and still contains all solution states.) The output of the labeling stage is fed to a pruner, which removes subtrees rooted at nodes labeled with known non-empty conflict sets. Again, demand-driven execution guarantees that the excised subtrees never need to be labeled. Because of this arrangement, the labeler is allowed to assume that if it labels a node with a non-empty conflict set, it will never be called on a descendent of that node, so it need not annotate such descendents properly; this allows simpler labeler code. After pruning, the solution nodes are just the leaves of the tree annotated with known empty conflict sets; the remainder of the pipeline simply filters these out.

The framework of Figure 6 is sufficiently general-purpose to accommodate all the search algorithms discussed in the remainder of the paper. By instantiating `search` with the labeler function `bt` we obtain a simple backtracking solver `btsolver` that behaves just like `btsolver0`. The more sophisticated algorithms discussed below are all obtained by using fancier labeler functions, leaving `search` itself unchanged.

6 Heuristics and Search Order

As with the naive solver, if we are interested in only the first solution rather than all solutions, we can still use `search` unchanged; we merely demand just the head of the solution list. Since solutions are always extracted in left-to-right order, this implies that the time required to find the first solution will be very sensitive to the order in which values are tried for each variable. The use of *value-ordering* heuristics is well-established in the imperative search literature. Such heuristics can be implemented using specialized generator functions that produce the initial tree in the desired order. A more modular approach, however, is to view these heuristics as as *rearrangements* of a canonically-ordered initial tree; this keeps the initial generator simple and allows multiple heuristics to be readily composed.

Such rearrangement heuristics can be easily expressed in our framework by incorporating them into the `labeler` function. For example, `queens` search can be speeded up by considering values in random order. The following function `hrandom` in Figure 7 transforms a canonical tree by randomizing its children (using a random number generator not shown here). The application `(btr` *seed*`)` returns a labeler that combines randomization with standard backtracking search. We have implemented a number of other such heuristics, both generic and problem-specific, but we omit details from this paper for lack of space.

```
type Table = [Row]       -- indexed by Var
type Row = [ConflictSet] -- indexed by Value

bm :: Labeler
bm csp = mapTree fst . lookupCache csp . cacheChecks csp (emptyTable csp)

emptyTable :: CSP -> Table
emptyTable CSP{vars,vals} = []:[[Unknown | m <- [1..vals]] | n <- [1..vars]]

cacheChecks :: CSP -> Table -> Transform State (State, Table)
cacheChecks csp tbl (Node s cs) =
  Node (s, tbl) (map (cacheChecks csp (fillTable s csp (tail tbl))) cs)

fillTable :: State -> CSP -> Table -> Table
fillTable [] csp tbl = tbl
fillTable ((var' := val'):as) CSP{vars,vals,rel} tbl =
  zipWith (zipWith f) tbl [[(var,val) | val <- [1..vals]] | var <- [var'+1..vars]]
    where f cs (var,val) =
            if cs == Unknown && not (rel (var' := val') (var := val))
            then Known [var',var]
            else cs

lookupCache :: CSP -> Transform (State, Table) ((State, ConflictSet), Table)
lookupCache csp t = mapTree f t
  where f ([], tbl)      = (([], Unknown), tbl)
        f (s@(a:_), tbl) = ((s, cs), tbl)
          where cs = if tableEntry == Unknown then checkComplete csp s else tableEntry
                tableEntry = (head tbl)!!(value a-1)
```

Figure 8: Backmarking

7 Backmarking

Given the formulation of backtracking search as a pipelined algorithm with separate labeling and pruning phases, using a tree annotated with conflict sets as intermediate data structure, it makes sense to ask if there are other ways to perform the labeling phase. `bt` works by checking each assignment against all previous assignments in its state. Although this approach checks the overall consistency of each partial state only once, it can still perform many duplicate pairwise consistency checks because all the children of a given node are isomorphic. Consider a node s at level l, and consider any descendent of s. In checking the consistency of the descendent, pairwise checks will be made between its assignment and all the assignments in s at levels less than l. These checks will be duplicated for the corresponding descendents of *every* sibling of s (unless, of course, they had an inconsistent ancestor and have been pruned away). For an example, compare the leftmost nodes of the left-most and right-most subtrees on level 5 of Figure 4: to generate these conflict sets, `bt` makes the same three comparisons in each case.

An alternative approach is to *cache* the results of such consistency checks so they can be reused for each sibling; this should reduce the total number of consistency checks at the cost of the space needed for the cache. Figure 8 shows a Haskell algorithm incorporating this idea. We annotate each node with a cache to store information about inconsistencies between that node's state and the assignments made in its descendents. Each cache is organized as a table of earliest conflict sets for *all* descendents, indexed by level (greater than or equal to the node's own level) and value; the table is represented as a list of lists. The root has a table in which every entry contains `Unknown`. `fillTable` computes the table contents for a node based on the node's assignment and the node's parent's table by considering each possible future assignment in turn. If the parent's table already records a known conflict pair for the future assignment, that conflict

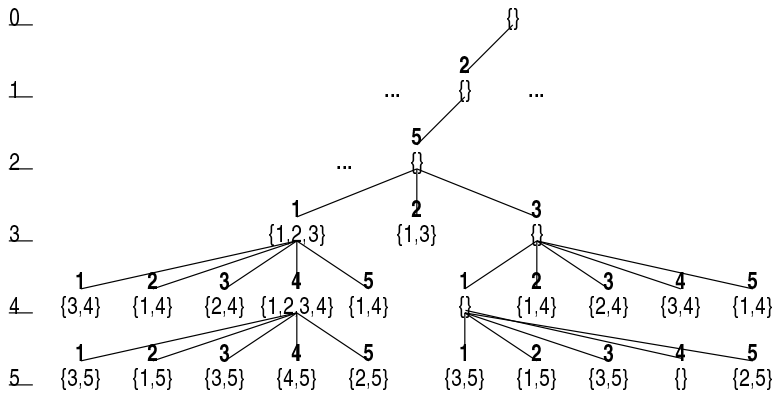


Figure 9: Same portion of search tree for `queens 5`, annotated with conflict sets as computed by `bj`.

pair is copied into the current table; otherwise a conflict check is performed and the result (a known conflict pair or `Unknown`) is recorded. Note that each node's table contains a refinement of the information in its parent's table, with a table at level l containing complete consistency information about assignments at level l. Once the tree has been annotated with cache tables, `lookupCache` is mapped over each node to extract the conflict pair for the node's own assignment from the node's table; if the node has no recorded conflicts and represents a complete state, it is a solution and is therefore given an empty conflict set. The ultimate annotated tree is identical to that produced by `bt`.

As usual, we rely on lazy evaluation to avoid building the tables or their contents unless they are needed. So most of the tables remain unbuilt, and the actual order in which consistency checks is performed is similar to `bt`. The important point is that, because many of a node's table entries are inherited from its parent's table, all duplicate consistency checks are avoided.

As before, we obtain a complete solver by using `bm` as the `labeler` parameter to `search`. Somewhat surprisingly, this algorithm turns out to be equivalent (in terms of consistency checks made) to a standard imperative algorithm called *backmarking*[1].

8 Conflict-Directed Backjumping

The `bt` and `bm` algorithms annotate inconsistent nodes with known conflict sets, but most internal nodes remain marked `Unknown`. If we could somehow compute non-empty conflict sets for internal nodes closer to the root of the tree, we could prune larger subtrees and so speed up search. In fact, many such nodes *do* have non-empty conflict sets; for example, see the leftmost node at level 3 in Figure 9.

One approach to computing internal node conflict sets is to construct them bottom-up from the conflict sets of a subset of their children. To do this, we make use of two key facts about conflict sets:

- (i) If a node s at level l has a child (at level $l+1$) with a known conflict set CS that does not contain $l+1$, then CS is also a conflict set for s. (In particular, if s has a child with an empty conflict set, then s also has an empty conflict set.)
- (ii) If all the children of node s at level l have non-empty conflict sets $CS_1, CS_2, \ldots, CS_n$, then $(CS_1 \cup CS_2 \cup \ldots \cup CS_n) \cap \{1, \ldots, l\}$ is a conflict set for s.

```
bjbt :: Labeler
bjbt csp = bj csp . bt csp

bj :: CSP -> Transform (State, ConflictSet) (State, ConflictSet)
bj csp = foldTree f
  where f (a, Known cs) chs = Node (a,Known cs) chs
        f (a, Unknown)  chs = Node (a,Known cs') chs
          where cs' = combine (map label chs) []

combine :: [(State, ConflictSet)] -> [Var] -> [Var]
combine []                acc = acc
combine ((s, Known cs):css) acc =
  if maxLevel s `notElem` cs then cs else combine css (cs `union` acc)

bj' :: CSP -> Transform (State, ConflictSet) (State, ConflictSet)
bj' csp = foldTree f
  where f (a, Known cs) chs = Node (a,Known cs) chs
        f (a, Unknown) chs =
            if knownConflict cs' then Node (a,cs') [] else Node (a,cs') chs
                where cs' = Known (combine (map label chs) [])
```

Figure 10: Conflict-directed backjumping.

These facts are easy to prove from the definition of conflict set. Intuitively, fact (i) says that if *any* child of s has conflicts that don't involve v_{l+1}, then *all* children of s have (at least) the same conflicts, and hence so does s itself. (The special case just says that if a child of s can be extended to a solution, then so can s.) Fact (ii) says that if no child of s can be extended to a solution, then neither can s, and any solution must differ from s in the value of at least one of the offending variables of one of the children. Fact (i) is the crucial one for optimizing search, since it permits the parent's conflict set to be computed from a strict *subset* of the children's conflict sets.

We can now define a lazy bottom-up algorithm for computing internal node conflict sets from a tree that has been (lazily) "seeded" with at least one conflict set along every path from root to leaf. Function `bj` in Figure 10 is a Haskell version of this labeling algorithm. At each parent node that doesn't already have a conflict set, `bj` calls `combine` to build one. `combine` inspects the conflict sets of the children in turn. If it finds a child to which fact (i) can be applied, it immediately returns this as the conflict set for the parent; if no such child is found, it applies fact (ii).[2] Under lazy evaluation, the subtrees corresponding to the remaining children are never explored.

This algorithm works correctly for *any* initial seeding of conflict sets, but it is most effective when the conflict sets are small and contain low-numbered variables, because this increases the number of levels for which fact (i) can be applied. This is why we use *earliest* inconsistent pairs to represent consistency conflicts. The combination of `bj` with `bt` is commonly referred to as *conflict-directed backjumping (CBJ)* (or just backjumping) in the literature and it is the cornerstone of many newly-developed algorithms [6]. In its usual imperative formulation this algorithm is notoriously difficult to understand or prove correct. While we have relied on the analysis of Caldwell, et al. [4] for our understanding of conflict sets, we are unaware of any description of the algorithm as a form of labeling.

While `search bjbt` behaves just like imperative CBJ in the sense that it performs the same number of consistency checks, it has an unfortunate space leak. The problem is that the pruning phase cannot remove the children of a node until that node's conflict set has been computed, but that computation may generate a substantial part of the children's subtrees into memory. We can plug the space leak effectively, if not too

[2] To simplify the implementation, we don't bother performing the intersection step in fact (ii), since it is harmless for a node's (non-empty) conflict set to include indices of its descendents.

```
fc :: Labeler
fc csp = domainWipeOut csp . lookupCache csp . cacheChecks csp (emptyTable csp)

collect :: [ConflictSet] -> [Var]
collect [] = []
collect (Known cs:css) = cs `union` (collect css)

domainWipeOut :: CSP -> Transform ((State, ConflictSet), Table) (State, ConflictSet)
domainWipeOut CSPvars t = mapTree f t
  where f ((as, cs), tbl) = (as, cs')
          where wipedDomains = ([vs | vs <- tbl, all (knownConflict) vs])
                cs' = if null wipedDomains then cs else Known(collect(head wipedDomains))
```

Figure 11: Forward checking.

neatly, by adding additional pruning into the labeler itself, as illustrated by `bj'`.

9 Forward Checking

Another way of assigning conflict sets to consistent internal nodes can be developed on the basis of the the cache tables introduced for backmarking (Section 7). Recall that these tables record, for each node, the earliest conflict sets for all descendent nodes; table entries for consistent nodes will remain marked `Unknown`. Suppose, however, that the table for some node n at level i contains a row, corresponding to a domain level $j > i$, in which every entry contains a non-empty conflict set. Then it is evident that the node can never be extended to a solution, because the assignments in n rules out all possible values for variable j. (As an example, consider the the left diagram in Figure 5; if we add a queen at position (4,4), then we can immediately see that no row placement will work for column 5.) Therefore, there must exist a non-empty conflict set for n. By labelling n with such a set, we can avoid further search in the subtree rooted at n. This technique has been called *domain wipeout* [1]. The combination of domain wipeout with backmarking corresponds to the well-known imperative algorithm called *forward checking*. Because our cache table construction is lazy, we have actually rediscovered ("for free") *minimal (*or *lazy) forward checking*, itself a recent discovery in the imperative literature [5].

Figure 11 shows code for implementing domain wipeout. To gather a list of `wipedDomains` and test whether it is non-empty is straightforward. The interesting question is what conflict set to assign to the node n if domain wipeout has occurred. Since it is always valid to throw additional variables into a non-empty conflict set, we could just use the set $\{1, \ldots, i\}$. But it is better to use the smallest available conflict sets based on the available information, because this can increase their utility for other algorithms (e.g., CBJ). In this case, the cache table row for a wiped-out domain records which existing assignment rules out each possible value for that domain. The union of the variables in these assignments (restricted to $\{1, \ldots, i\}$)[3] is a valid conflict set for n, since any solution must assign a different value to at least one of them. If there is more than one wiped out domain, we could compute a conflict set from any one of them; for simplicity and to limit computation, `domainWipeOut` just chooses the first.

10 Mixing and Matching

A major advantage of our declarative approach is that we can trivially combine algorithms using function composition, so long as they take a consistent view of conflict set annotations. The combination of forward

[3] Again, we simplify the implementation by omitting the restriction step, which is harmless.

Queens	8	9	10	11	12	13
CSPlib BT	0.01	0.05	0.27	1.50	8.91	57.34
ghc monolithic BT	0.14	0.60	3.20	18.03	108.34	686.92
ghc `btsolver0`	0.56	2.84	14.18	76.29	440.72	2686.13

Table 1: Runtime in seconds for different versions of simple backtracking search for the n-queens problem.

checking and backjumping

```
bjfc csp = bj csp .  fc csp
```

is well known, although to our knowledge it has not previously been achieved for lazy forward checking. Imperative forward checking is traditionally described as filtering out all the conflicting values from the domains of future variables; this makes it hard to explain how it can be profitably combined with backjumping, since the latter would seem to have no information on which to base backjumping decisions. Our viewpoint is that forward checking is just a more (time-)efficient way of generating conflict sets, which makes the combination perfectly reasonable.

Similarly, the combination of backmarking and backjumping

```
bjbm csp = bj csp .  bm csp
```

is tricky to implement correctly in an imperative setting [11], but is simple for us, and turns out to do do fewer consistency checks on `queens` than any of our other algorithms.

Once problem-specific value ordering heuristics are introduced, many more possibilities for new algorithm design open up. Since the best combination of algorithm features tends to depend on the particular problem at hand, it is important to be able to experiment with different combinations; our framework makes this extremely easy.

11 Experimental Results

To estimate the cost of modularity and laziness we wrote an integrated, strict version of simple backtracking search for the n-queens problem in Haskell and compared the runtime with that of `btsolver0`. Table 1 reports the results; they indicate an overhead factor of about four times. The measurements were taken using `ghc` (the Glasgow Haskell compiler) version 3.02 with optimization turned on, running on a lightly loaded Sun Ultra 1 under Solaris 2.5.1. We also show the runtime of an optimized C library for solving CSPs [17] compiled with `egcs` version 2.93.06 using `-O4` on the same platform; it runs an order of magnitude faster, partly because it performs consistency checks via lookup into a precomputed table. Table 2 gives the number of consistency checks made by the different algorithms for the n-queens problem.

12 Related Work

Hughes [9] gives a lazy development of minimax tree search. Bird and Wadler [3] treat the n-queens problem using generate-and-test and lazy lists. Laziness (not in the context of lazy languages) has been used for improving the efficiency of existing CSP algorithms [15, 5], but as far as we know laziness has not been previously been used to modularize any of the CSP algorithms presented here.

Queens	5	6	7	8	9	10	11	12	13
`bjbm`	276	909	3158	11928	49369	210210	975198	4938324	26709008
`bjfc`	279	916	3182	12229	51314	218907	1026826	5231284	28387767
`bm`	276	944	3236	12308	50866	220052	1026576	5224512	28405086
`fc`	279	920	3189	12276	51642	220745	1038129	5297651	28817439
`bjbt`	405	1828	8230	41128	214510	1099796	6129447	36890689	233851850
`bt`	405	2016	9297	46752	243009	1297558	7416541	45396914	292182579
Solutions	10	4	40	92	352	724	2680	14200	73712

Table 2: Number of consistency checks performed by various algorithms on the n-queens problem. Algorithms are identified by their labeler function name.

Many reformulations of standard algorithms into a framework exist in the literature [8, 6, 16, 2], but the frameworks typically aren't modular; in the best case the differences between two algorithms are highlighted by showing which lines of pseudo-code have changed [11]. Algorithms have been classified according to the amount of arc consistency (AC) they do [12] or the number of nodes visited [11]. These classifications have shown that the backmarking and forward checking algorithms, which were previously thought of as being fundamentally different, actually share the same foundation [1], as we independently rediscovered (Section 9). There often remains confusion, even among experts in the field, about which algorithm a given description really implements.

Considering how long the standard algorithms have existed and how much they are used, there have been surprisingly few proofs of correctness. A correctness criterion for search algorithms based on soundness and completeness was presented in Kondrak [11] and an automatic theorem prover was used to derive the algorithms in Caldwell, et al. [4].

The term "conflict set" is very common in the literature, but a precise definition is difficult to achieve; we base ours on that of Caldwell, et al. [4].

13 Conclusion

Expressing algorithms in a lazy functional language often clarifies what an algorithm does and what invariants it depends on. With a little bit of care we can modularize code that traditionally has been expressed in monolithic imperative form. Experimentation is also very easy. New combinations of algorithms, such as forward checking plus conflict-directed backjumping, can be expressed in a single line of code; the equivalent algorithm in the imperative literature requires many lines of (mysterious) C or pseudocode. Despite the overheads introduced by laziness and use of Haskell, large experiments can be conducted. For example, combining `hrandom` with `bjbt` allowed us to find solutions for the queens problem with well over 100 queens, even using the Haskell interpreter Hugs.

The major problem of working with lazy code is difficulty in predicting runtime behavior, particularly for space. Very minor code changes can often lead to asymptotic differences in space requirements, and the available tools for investigating such problems in Haskell are inadequate.

For future work, we plan to work on formal proofs of algorithmic correctness, which should be relatively easy in our framework, and to investigate variable-reordering heuristics, which are at the core of current work in the AI search literature.

References

[1] F. Bacchus and A. Grove. On the forward checking algorithm. In *Principles and Practice of Constraint Programming*, pages 293–309, Cassis, France, September 1995.

[2] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming*, pages 258–275, Cassis, France, September 1995.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[4] J. L. Caldwell, I. P. Gent, and J. Underwood. Search algorithms in type theory. In *Submitted to: Theoretical Computer Science: Special Issue on Proof Search in Type-theoretic Languages*, September 1997.

[5] M. J. Dent and R. Mercer. Minimal forward checking. In *Prec. of the Int'l Conference on Tools with Artificial Intelligence*, pages 432–438, New Orleans, Louisiana, 1994. IEEE Computer Society.

[6] D. H. Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems*. PhD thesis, University of California Irvine, 1997.

[7] A. Gill, J. Launchbury, and S. Peyton Jones. A short-cut to deforestation. In *Proc. ACM FPCA*, 1993.

[8] M. L. Ginsberg. Dynamic backtracking. *Journal of Artifical Intelligence Research*, 1:25–46, 1993.

[9] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[10] D. King and J. Launchbury. Structuring depth first search algorithms in Haskell. In *Proc. ACM Principles of Programming Languages*, 1995.

[11] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Master's thesis, University of Alberta, 1994.

[12] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[13] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5(3):16–23, June 1990.

[14] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[15] T. Schiex, J. C. Regin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proc. of AAAI*, pages 216–221, Portland, Oregon, USA, 1996.

[16] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 1993.

[17] P. van Beek. A 'C' library of constraint satisfaction techniques., 1999. Available from `ftp://ftp.cs.ualberta.ca/pub/vanbeek/software/`.

Persistent Triangulations*

Guy Blelloch Hal Burch Karl Crary Robert Harper Gary Miller Noel Walkington

Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Triangulations of a surface are of fundamental importance in computational geometry, engineering simulation, and computer graphics. For example, the convex hull of a set of points may be constructed as a triangulation, and there is a close relationship between Delaunay triangulations and Voronoi diagrams in geometry. Triangulations are ordinarily represented as mutable graph structures for which both edge traversal and adding edges take constant time per operation. These representations of triangulations make it difficult to support *persistence*, including "multiple futures", the ability to use a data structure in several unrelated ways in a given computation, "time travel", the ability to move freely among versions of a data structure, or parallel computation, the ability to operate concurrently on a data structure without interference. We propose a new representation of triangulations that supports persistence. To demonstrate its use we give a new algorithm for the three-dimensional convex hull that is asymptotically optimal in the expected case, and we give an implementation of a terrain-modelling algorithm based on this representation. To assess its practicality we measure the performance of both applications.

1 Introduction

A *persistent* data structure is one for whose operations—even those that "update" or "modify" it—preserve the data structure across calls. To achieve persistence, update operations must create a "fresh" copy of the data structure so that the original is not disturbed by the operation. Persistent data structures arise naturally in "value-oriented" programming languages such as ML or Haskell. In these languages all data structures are values that are passed as arguments and returned as results, much as numbers are handled in nearly every language.

In contrast, most familiar data structures are *ephemeral*—the operations on the data structure "mutate" it by modifying its representation in memory in such a way that all references to it change simultaneously. Ephemeral data structures arise naturally in "object-oriented" programming languages, including Java or C++. In these languages data structures are thought of as regions of mutable storage that is modified by the operations on the structure.

Persistent data structures offer a number of advantages not shared by their ephemeral counterparts. The term persistence, however, is sometimes use loosely in the literature and there are several types of persistence that can be categorized by the features they support. The key features we are concerned with are:

1. *time travel:* the ability to go back and view any previous version of a data structure,

*This research was supported in part by NSF Grant CCR-9706572. Burch was supported by an NSF Graduate Fellowship.

2. *multiple futures*, the ability to go back and make a sequence of modifications to a previous version of the data structure without affecting the current version (this allows for a version tree),

3. *combining*, the ability to combine persistent data structures, such as taking the union of two persistent sets, and

4. *implicit parallelism*, the ability to view and modify different versions of a data structure in parallel.

Driscoll, *et. al.* [10] distinguish between *partially persistent* and *fully persistent* data structures. The former support only time travel, whereas the latter also support multiple futures. They describe techniques for building both partially and fully persistent data structures, but their methods do not support combining or implicit parallelism. Driscoll, Sleator and Tarjan [11] introduced the term *confluently persistent* for fully persistent structures that also support combining, and showed a technique for supporting *confluently persistent* lists with catenation. Their technique does not support parallelism. We will use the term *strongly persistent* for a data structure that supports all four features.

The above mentioned techniques are heavily based on the use of side-effects. It is well known, however, that purely functional programs (no side-effects) are inherently persistent, and support at least time-travel, multiple-futures and combining. Furthermore, if the program is *strictly functional* (*i.e.*, does not use lazy evaluation or any other form of memoization) then it will also support implicit parallelism. Tarjan and Kaplan [18] make use of this in a design of a strictly functional catenable list that is strongly persistent. Okasaki [20] developed a simpler algorithm with similar time bounds based on functional programs. Since his method uses lazy evaluation, however, it does not support parallelism.

The subject of this paper is the persistent representation of *closed surfaces* in multi-dimensional space. Closed surfaces are of fundamental importance in computational geometry, and have a number of applications in a wide variety of areas, including geographic information systems, mesh generation for engineering simulations, and surface representations in computer graphics. One application of closed surfaces is the construction of the convex hull of a set of points in three dimensions. The convex hull of a set of points is the surface, or boundary, of the smallest convex polytope containing those points. Another application is to terrain modelling, in which the topography of a geographical region is approximated to within a specified resolution by a closed surface.

A number of representations of closed surfaces have been considered in the literature [16, 4, 8, 7], but all are based on ephemeral data structures such as graphs or mutable dictionaries. Using these ephemeral data structures, several asymptotically optimal algorithms for the construction of the convex hull of a set of points in three dimensions are known [4, 7]. Garland and Heckbert's terrain modelling algorithm [13] is also based on an ephemeral representation of surfaces.

Persistence offers a number of advantages over the more familiar ephemeral representations. In the case of the convex hull algorithm, we may exploit implicit parallelism to provide a simultaneous display of the construction of the hull during its construction, without imposing any synchronization or checkpointing overhead. By exploiting time travel, we may move backwards and forwards among stages of the construction of the hull, allowing the user to explore the dynamics of its creation or to modify the unconsidered points to see the effect on the final hull. In the case of terrain modelling, persistence supports viewing the model at many different resolutions, on demand. Although we have not explored the direct use of persistent surfaces in algorithm design, persistent data-structures are used as components of several algorithms in other areas of computational geometry [21, 15, 17].

Conventional implementations of the 3-dimensional convex hull algorithm rely on mutable data structures such as the doubly-connected edge list described by de Berg, *et. al.* [4]. The hull is represented by a planar graph whose nodes are triangles and whose edges represent the adjacency relation among them. The analysis of these algorithms relies on the assumption that graph edges may be traversed, and new edges added, in

constant time.[1] A naïve translation of these algorithms to the persistent setting would proceed by simply replacing the ephemeral graph structure by a persistent dictionary recording the adjacency relation. Using a simple dictionary representation the cost of the fundamental graph operations increases from $O(1)$ to $O(\lg n)$, which would lead to a sub-optimal $O(n \lg^2 n)$ time bound in the persistent case.

One approach to reducing this cost might be to use a persistent representation of graphs (such as the one given by Erwig [12]) that performs well in the case that each graph has only one "logical future". The obvious disadvantage of this approach is that in the multiple-future case the performance once again degrades to sub-optimal. Moreover, these methods usually rely on so-called "benign" effects that inhibit parallelism. Another approach might be to use Dietz's persistent representation of arrays [9] for which fully persistent array updates and searches require only $O(\lg \lg n)$ time. Used naïvely in a convex hull algorithm Dietz's method would lead to a suboptimal time bound of $O(n \lg n \lg \lg n)$. Moreover, Dietz's representation does not support combining or implicit parallelism.

We therefore consider whether it is possible to achieve the $\Omega(n \lg n)$ lower bound while retaining the advantages of strong persistence. We present a new randomized algorithm, called the *bulldozer algorithm*, for the construction of the convex hull of a set of points in three dimensions that achieves the lower bound in the expected case. The crucial feature of the algorithm is that the expected number of adjacency relationship checks and updates among faces of the hull is at most $O(n)$, while the expected number of floating-point operations to determine the spatial relationships among points is $O(n \lg n)$. Since the algorithm only requires $O(n)$ operations on the surface, we can afford to spend $O(\lg n)$ time per operation without affecting the overall time bound. We can therefore use a simple dictionary based on balanced trees to represent the adjacency relationship of the surface.

Asymptotic complexity is important, but so are empirical performance measurements. To assess the practicality of our representation, we measure the performance of the hull algorithm and the terrain modelling algorithm on representative data sets. There are several types of experiment we could run. Rather than just measuring running times, which are strongly influenced by the machine used and the optimization of the algorithm, we counted the number of basic operations. For the numerical component of the algorithm we counted the number of floating point operations. For the manipulation of the surface we counted the number of key comparisons that are made in our dictionary implementation of the surface, which is based on Red-Black trees. For the convex-hull algorithms we expect both these counts to grow as $O(n \log n)$, but we were interested in comparing the constant factors. Our results are given in Section 4 and show that there are more floating-point operations than key comparisons over a wide variety of point distributions and sizes. Similar results for the terrain-modeling code are given in Section 5.

Since one could argue that a "key comparison" and associated overhead could be much greater than the cost of a floating-point operation, we also ran a timed experiment. To avoid biasing the results, our comparisons are made to the fastest 3D convex-hull algorithm we knew of [2], which was developed at the Minnesota geometry center. We first measured the time for Minnesota Quickhull using its own emphemeral surface representation. We then measured the additional time required to implement the surface operations using a dictionary instead of the ephemeral structure. Our initial experiments show that the dictionary adds a cost of about 50% to the overall cost of the algorithm.

2 Surfaces

Both the convex hull algorithm and the terrain modelling algorithm presented in later sections construct a two dimensional surface enclosing a set of points. In the case of the convex hull, this is the surface of the smallest enclosing convex polytope of a set of points, and in the case of the terrain modelling, the surface represents

[1] This may not be a reasonable assumption when the number of nodes is extremely large, due to memory hierarchy effects. Nevertheless, it is a standard assumption to ignore such issues.

an approximation to the topography of a geographical region. In both cases it is convenient to think of the surface as a connected set of triangles covering the surface; if the surface is specified by polygonal faces they are subdivided into triangles. Consequently, the surfaces of interest are sometimes called *triangulated surfaces*, or simply *triangulations*.

Following Giblin [14], we define a *closed surface* to consist of a set of triangles satisfying the following three conditions:

1. Any two triangles have at most one vertex or one edge (and its two vertices) in common; no other forms of overlap are permitted. This is called the *intersection condition*.

2. The surface is *connected* in the sense that there is a path from any vertex to any other vertex consisting of edges of the triangles of the surface.

3. The set of edges "opposite" any vertex, called the *link* of that vertex, forms a simple, closed polygon.

This definition relies on the familiar concept of a triangle. A triangle consists of a set of three distinct vertices, specified in some order. This raises the question of when two triangles are equivalent. Under an *ordered* interpretation, $\triangle ABC$ is distinct from both $\triangle BCA$ and $\triangle CAB$, even though they enumerate the vertices in the same sequence, and is also distinct from $\triangle ACB$, which reverses the order of presentation. Two orderings that differ by an even permutation (*i.e.*, that can be obtained from one another by an even number of swaps) are said to determine the same *orientation*. Thus $\triangle ABC$, $\triangle BCA$, and $\triangle CAB$ all have the same orientation, whereas $\triangle ACB$ (and its even permutations) have the opposite orientation. The orientation may be thought of as determining two "sides" of a triangle; $\triangle ABC$ is the "front" of $\triangle ABC$, and, correspondingly, $\triangle ACB$ is the "back" of $\triangle ABC$. Under an *oriented* interpretation we identify triangles that have the same orientation, and distinguish those that do not.

Following Giblin, we maintain a careful distinction between the configuration of the triangles on the surface (*i.e.*, their adjacency relationships) and the embedding of the triangles in three-dimensional space (*i.e.*, the assignment of coordinates to their vertices). When embedding a triangle $\triangle ABC$ in three-dimensional space, we require that the points assigned to the vertices be *affinely independent*, which is to say that the vectors $B - A$ and $C - A$ are linearly independent, or, equivalently, that the three points are not collinear. The convex hull algorithm will determine not only the configuration of triangles, but also their embedding in three-dimensional space.[2]

A closed surface is a special case of the more general concept of a *simplicial complex* [14, 1], which applies in an arbitrary dimension. Our implementation of the three-dimensional convex hull and of the terrain modelling algorithm are based on an abstract type of simplicial complexes. Not only does this support generalization to higher-dimensional spaces, but it also allows us to experiment with various implementations of them without disturbing the application code. Indeed, we experimented with several different implementations before settling on the one we describe here.

Just as a closed surface is a set of triangles satisfying some conditions, a simplicial complex is a set of *simplices* over a set of *vertices* satisfying some related conditions. A zero-dimensional simplex is a "bare" vertex, a one-dimensional simplex is a line segment, a two-dimensional simplex is a triangle, a three-dimensional simplex is a tetrahedron, and so on. A complex is a configuration of simplices subject to some simple conditions that ensure that the simplices "fit together" to form a coherent "solid" in n-dimensional space.

We assume given a totally ordered set V of *vertices*.[3] An *n-dimensional ordered simplex*, or *n-simplex*, is an $(n + 1)$-tuple of distinct vertices. An ordered simplex is *oriented* iff we do not distinguish between two orderings that differ by an even permutation (one that can be expressed as an even number of swaps). A simplex s is a *sub-simplex*, or a *face*, of a simplex t, written $s \leq t$, iff s is a subsequence of t.

[2] To avoid degeneracies and to simplify the presentation, we assume that the input set of points to the hull algorithm has the property that no four points are coplanar.

[3] This is not ordinarily required in the mathematical setting, but is necessary for implementation reasons.

```
signature VERTEX =
  sig
    type vertex
    val compare : vertex * vertex -> order
    type point
    val new : point -> vertex
    val loc : vertex -> point
  end
```

Figure 1: Signature of Vertices

An *n-dimensional, oriented, pure simplicial complex*, or just *n-complex* for short, consists of a set V of vertices and a set S of oriented simplices satisfying the following conditions:

1. Every vertex determines a 0-simplex. We usually do not distinguish between a vertex v and its associated 0-simplex (v).
2. Every sub-simplex of a simplex in K is also a simplex of K.
3. Every simplex $s \in S$ is a sub-simplex of some n-simplex in S. That is, there are no m-simplices, with $m < n$, other than those that are faces of an n-simplex in S.

A *closed surface* is a 2-complex in which the link of every 0-simplex is a simple, closed polygon having that 0-simplex as an interior point.

The signature (interface) of the simplicial complex abstract type is given in Figure 3. This abstraction relies on an abstract type of vertices, whose signature is given in Figure 1, and an abstract type of simplices, whose signature is given in Figure 2. Taken together, these signatures summarize the entire suite of operations available to applications that build and manipulate complexes. To fix ideas we summarize the operations provided by these abstractions.

The signature `VERTEX` specifies that vertices admit a total ordering, which is required for efficiently associating data with vertices. In particular we associate a point with each vertex; this is used to embed a simplex in space, as described earlier. The embedding is established by the `new` operation, which creates a "new" vertex at the specified point. The location of a vertex in space is obtained using the `loc` operation, which yields the point in space associated with vertex. The type of points is left completely unspecified since the simplicial complex package need not be concerned with its exact representation.

The signature `SIMPLEX` defines the abstract type of (ordered) simplices over a given type of vertices. As with vertices, we require that simplices be totally ordered by some unspecified order relation so that simplices may be used as keys in a dictionary. The operation `dim` yields the dimension of a simplex. Since the order of vertices in a simplex is significant, we distinguish one vertex as the *apex* of the simplex, with the others following in order; this is the first vertex in the enumeration of vertices of the simplex. The `vertices` operation yields the sequence of vertices of a simplex in order, apex first.[4] An n-simplex is created by applying the `simplex` operation to a sequence of $n + 1$ vertices; the first vertex in the sequence is the apex. The `orders` operation yields a sequence of $n + 1$ orderings of the simplex with the same orientation, one ordering for each choice of apex. The `faces` operation yields a sequence of $(n - 1)$-dimensional subsimplices of a given n-simplex. The `flip` operation inverts the orientation of a simplex (flips to its reverse side). The `down` operation passes from an n-simplex to its apex and the "opposing" $(n - 1)$-simplex of that

[4] We make use of an abstract type of sequences, a form of immutable array whose primitive operations are designed to support implicit parallelism [5].

```
signature SIMPLEX =
  sig
    structure Vertex : VERTEX
    type simplex
    val compare : simplex * simplex -> order
    val dim : simplex -> int
    val vertices : simplex -> Vertex.vertex seq
    val simplex : Vertex.vertex seq -> simplex
    val down : simplex -> Vertex.vertex * simplex
    val join : Vertex.vertex * simplex -> simplex
    val orders : simplex -> simplex seq
    val faces : simplex -> simplex seq
    val flip : simplex -> simplex
  end
```

Figure 2: Signature of Simplices

apex. (In the case of a triangle, this is the base opposite to a specified vertex of the triangle.) The `join` operation builds an n-simplex from a given vertex and $(n-1)$-simplex, taking the vertex as apex and the $(n-1)$ simplex as its opposite face.

The signature `SIMPCOMP` specifies the abstract type of simplicial complexes. There are no mutation operations on complexes. Instead we supply operations to create new complexes from old, as discussed in the introduction. The type `'a complex` of n-dimensional simplicial complexes is parameterized by a type `'a` of data values associated with the n-simplices of the complex. The dimension of the complex is a fixed property of the abstract type; different instances of the abstraction may have different dimension. The empty complex is the value `empty`; the operation `isempty` tests for it. The sequence of vertices of a complex are returned by the `vertices` operation, in an arbitrary order. The simplices of a given dimension (at most `dim`) are returned by the `simplices` operation. The `grep` operation finds all the simplices of maximal dimension having a given simplex as a face. More precisely, given a dimension $d \leq$ `dim` and a d-simplex s, `grep` returns the sequence (in unspecified order) of simplices of dimension `dim` having s as a face. The `find` operation is a specialization of `grep` for dimension `dim` -1. The operation `add` adds a simplex to a complex, with specified data value; to ensure that the condition 3 in the definition of simplices is preserved, we may only add an n-simplex to an n-complex. The operation `rem` removes a simplex from a complex, yielding the reduced complex. The `update` operation applies a specified function to the data values of every simplex in the complex, yielding a new complex.

In our implementation, an n-simplex is represented by a sequence of vertices of length $n+1$, with the apex being the lead vertex of the sequence. The `down` operation strips off the apex and returns the remaining $(n-1)$-simplex, as described above. Simplices are compared by comparison of sequences so that different orderings determine different simplices. We implement complexes using the `Map` signature taken from the SML/NJ library. An $n > 1$ complex is represented by a mapping from vertices to the set of $(n-1)$-complexes incident on it. (In the case $n = 2$, each vertex has associated with it the edges, together with their vertices, incident on that vertex.) A 1-complex is implemented specially to avoid the overhead of maintaining the map.

We may build an n-complex by a sequence of $n-1$ applications of a "bootstrapping functor" that builds an n-complex from an $(n-1)$-complex, starting with the direct implementation of the 1-complex. However, for reasons of efficiency, we choose to implement the 2-complexes directly, rather than by bootstrapping. In this optimized implementation we use the first vertex of a simplex as a key into a red-black tree [3]. Each node of the red-black tree then stores as its value an association list that maps the second vertex to the third

```
signature SIMPCOMP =
  sig
    structure Simplex : SIMPLEX
    type 'a complex
    val dim : int
    val empty : 'a complex
    val isempty : 'a complex -> bool
    val vertices : 'a complex -> Simplex.Vertex.vertex seq
    val simplices : 'a complex -> int -> Simplex.simplex seq
    val data : 'a complex * Simplex.simplex -> 'a option
    val grep : 'a complex -> int * Simplex.simplex -> Simplex.simplex seq
    val find : 'a complex * Simplex.simplex -> Simplex.simplex option
    val add : 'a complex * Simplex.simplex * 'a -> 'a complex
    val rem : 'a complex * Simplex.simplex -> 'a complex
    val update : 'a complex * Simplex.simplex * ('a -> 'a) -> 'a complex
  end
```

Figure 3: Signature of Simplicial Complexes

vertex and the data. Using an association list is adequate in practice since the number of entries is small (the average number is 6). To make the implementation optimal in theory one could convert to a balanced tree if the size of the list becomes too long.

In our direct implementation searching for a simplex involves searching the red-black tree and then the association list. Adding a simplex involves searching the red-black tree to see if the vertex is already there. If it is, the simplex is added to the existing association list, otherwise a new association list is created. We note that when a simplex is added, it needs to be added to the tree in all three orders that have the same orientation. Deleting a simplex involves searching the tree and deleting the simplex from the corresponding association list. If the association list becomes empty, then the tree node is also deleted. As with adding, the deletion needs to be executed in all three orderings.

3 Convex Hull: The Bulldozer Algorithm

It is well known that the problem of constructing the convex hull of a set of points in three dimensions requires $\Omega(n \lg n)$ time [4]. Asymptotically optimal algorithms for the problem are also known [8, 7] for the ephemeral case. In this section we will give a randomized optimal algorithm for the persistent case.

We will be concerned with *incremental* methods that extend the convex hull of a set of points to include a new point. Many algorithms, including our own, are based on *tent construction*. Given a point p exterior to the hull of a set of points, we may extend the hull to include this point as follows. We view the exterior point as a *light source* illuminating a subset of the faces of the hull. The boundary of the lit faces is a set of edges, which we call the *horizon*. We then construct a pyramidal *tent* whose apex is the exterior point and whose base is the horizon, removing the lit faces. This construction extends the convex hull to include the given point as a new vertex.

Several incremental algorithms based on the tent construction are known; they differ in how the exterior point is chosen, and how the set of exterior points is maintained during the construction. Our algorithm maintains, for each exterior point, one face that is visible to that point. In particular, the algorithm begins by selecting a point that will always be interior to the hull—we call this the *center point*. Consider the ray

from the center point to each exterior point. Each such ray penetrates a face, to which we associate the point. A face is visible to each of its associated points. The set of faces that are visible to any one point is connected, so knowing one visible face allows one to walk through the visible faces to find them all. Clarkson and Shor's algorithm [8], the Minnesota Quickhull algorithm [2], the algorithm presented by Motwani and Raghavan [19], our basic algorithm, and the Bulldozer algorithm [6] all maintain such information. All the other algorithms, however, either do not care which visible face the point is associated with or keep a complete list of visible faces for each point.

Our algorithm requires a representation of the hull, for which we use a simplicial complex, and some method for associating points with faces, for which we use the data associated with each simplex in the complex.

Each incremental step of our algorithm selects a random face that has points associated with it. A random point associated with this face is designated as the light source. The algorithm then finds all the other faces visible to the light source by searching adjacent triangles on the surface starting with the selected face. The algorithm defines a directed acyclic graph whose nodes are the visible faces and the horizon edges, and whose arcs connect adjacent faces or a face with one of its horizon edges. The selected face has in-degree zero (*i.e.*, it is the root), and the horizon edges have out-degree zero. The faces are visited in a topological ordering of the graph. When visiting a face, every point assigned to the face is either discarded, because it is interior to the hull, or pushed out along an out-going arc (hence the name "bulldozer" algorithm). This requires at most two plane-side tests per point. When the search is complete each point associated with any of the visible faces has either been discarded or associated with a horizon edge. One additional test can determine whether a point is interior to the hull or visible to the face formed by this edge and the light source.

This algorithm visits each visible face once. It is possible to show, by backwards analysis and Euler's formula, that the expected number of faces visited is linear in the number of input points when summed across all steps. The cost of the algorithm can be separated into plane-side tests and the cost of the graph traversal and surface manipulation (*i.e.*, finding adjacent faces). Each visit requires at most a constant number of graph and surface operations, each taking $O(\log n)$ time, hence the total expected cost of graph traversal and surface manipulation is $O(n \lg n)$. It is also possible to show that the expected number of plane-side tests is $O(n \lg n)$ [6]. Thus the total expected cost is $O(n \lg n)$.

4 Convex Hull: Experimental Evaluation

Although our theory shows using a purely persistent dictionary for storing a simplicial complex is asymptotically optimal, we are interested in the actual overhead. In particular we were worried that the constant factors could make the ideas impractical. For this reason we ran several experiments to study the overhead. These experiments involved measurements on the bulldozer 3d hull algorithm, and on a terrain triangulation algorithm, described in the next section. The goal in the experiments is to compare the work needed to maintain the simplicial complex to the other work in the algorithm. This other work mostly consists of the numerical aspects and is dominated by floating-point operations.

In our experiments we used the following 5 distributions of points in 3d:

1. **OnSphere**: Random uniformly distributed points on the unit 2-sphere (*i.e.*, the surface of the unit ball in 3d).

2. **EqHeavy**: Random points on the sphere that are weighted to be mostly on the equator. These are generated by producing random points on the sphere, stretching the equator (x and y coordinates) by a factor of 100 so that the distribution is on a disk like surface, and then projecting the points back down onto a sphere by scaling their length to one.

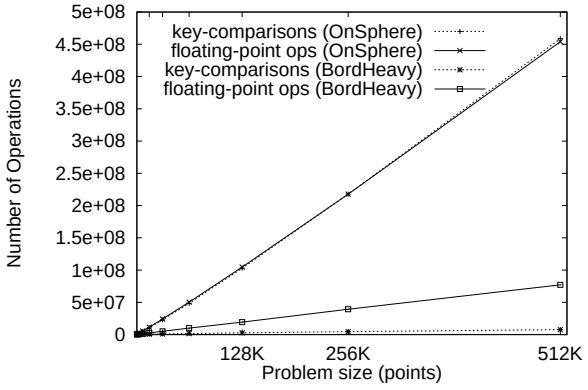


Figure 4: Operation counts as a function of input size for two of the distributions using the Bulldozer algorithm. The two sets of counts for OnSphere are almost identical.

3. **PolHeavy**: Random points on the sphere that are weighted to be mostly at the poles. These are generated by producing random points on the sphere, stretching the poles (z coordinate) by a factor of 100 so that the distribution is on a stretched ellipsoid surface, and then scaling the points back down onto a sphere as in the EqHeavy distribution.

4. **InBall**: Random uniformly distributed points in the unit ball.

5. **BordHeavy**: Generated by producing points randomly in a unit ball and then mapping each point (x, y, z) to the point $(x, y, x^2 + y^2 + z^2)$. It can be shown that using this distribution for n points, the size of the convex hull is expected to be $\Theta(n^{2/3})$.

We selected these since we wanted data sets both where all the points are in the final result (the expensive case) and where some are inside. We also wanted nonuniform distributions, which are what EqHeavy, PolHeavy, and BordHeavy give us.

To get a machine- and language-independent measurement of the costs we first measured various operation counts. For the manipulation of the simplicial complex (the topological part of the algorithm) we count both the number of dictionary operations and the total number of key-comparisons made by the dictionary code. For the numerical (geometric) part of the algorithm we count the number of plane-side tests, from which we can easily determine the number of floating-point operations.

As mentioned in Section 2, the simplicial complex is implemented using red-black trees with vertex identifiers used as keys. For a tree of size n each insertion, deletion or search will traverse $O(\lg n)$ nodes At each node, the key being searched (an integer identifier for the vertex) is compared to the key at the node. In addition to the key-comparisons made in the red-black tree, which are based on the first vertex of the simplex being searched, key-comparisons are also required when searching for the second vertex of the simplex in the association-list of the node that is found (see Section 2). Our key-comparison counts include these association-list comparisons. The *key-comparisons* is therefore a measure of the total number of red-black-tree nodes visited, plus the total number of association-list elements visited. Our theory states that the expected total number of dictionary operations is $O(n)$ and since the red-black tree operations visit $O(\lg n)$ nodes, the total number of expected key-comparisons is $O(n \lg n)$.

We measured the number of key-comparisons and floating-point operations for all the distributions and for a range of input sizes up to 512K points. A graph showing the operation counts as a function of size

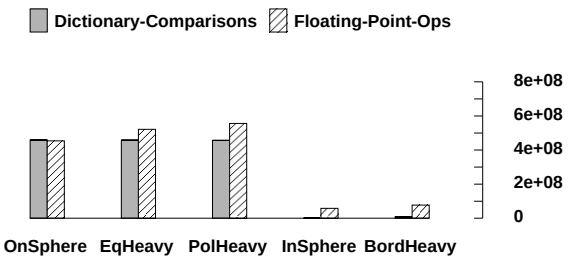


Figure 5: Operation counts for all five data distributions using the Bulldozer algorithm. The input size is 512K points.

is given in Figure 4 for two of the distributions. A bar graph showing the operation counts for the five distributions on 512K points is given in Figure 5. The graphs show that the number of key-comparisons is approximately the same as the number of floating-point operations for the first three distributions in which all the points are on the sphere. For the other two distributions in which some points are inside the ball, the number of key-comparisons is very much less than the number of floating-point operations (by a factor of 30 for the InBall distribution and a factor of 10 for the BordHeavy distribution). This is to be expected since the resulting hull is significantly smaller than the size of the input, and the simplicial-complex operations are only used on the simplexes that are actually created, while plane-side tests are required on all the input points.

We were also interested in actual running time of the simplicial complex code since one might imagine that traversing a node of a tree is more expensive than a floating-point operation. To be fair on this measure we wanted to compare times to a well tuned existing implementation of 3D Convex Hull. We therefore selected the Minnesota Quickhull code [2]. Since our code is written in ML and the Minnesota code is written in C, we could not compare the times directly. We also did not want to completely rewrite our code in C, or the Minnesota code in ML. Instead we instrumented our code to dump out traces of all the operations on the simplicial complex. We then wrote C code that simulates the complex operations using balanced trees and linked lists. The idea is to get an sense of how much time relative to the Quickhull code the persistent implementation of the simplicial complex requires. The results are shown in Figure 6. As can be seen, the cost of the simplicial-complex operations is at most half the total cost of the Minnesota code, and this is for a distribution where the number of operations on the complex is high. Since some of the cost of the Minnesota code is dedicated to manipulating its representation of the simplicial complex (it would be hard actually to separate this out) it is reasonably safe to conclude that using a persistent dictionary in their code for manipulate the surface would incur less than a 50% overhead, and for many distributions very much less.

5 Terrain Modeling: Experimental Evaluation

One interesting real-world application of the convex hull algorithm is to terrain modeling [13]. Terrain data is important to many real-world applications, such as flight simulators. However, rendering a terrain at full resolution is impractical for terrains of any significant size. Therefore, applications that rely on terrain data require terrain models that approximate full terrains using substantially fewer polygons.

Given a two-dimensional array of evenly spaced height samples from the full terrain, a terrain modeling procedure computes a triangulation of the terrain that minimizes the error between the actual sample values

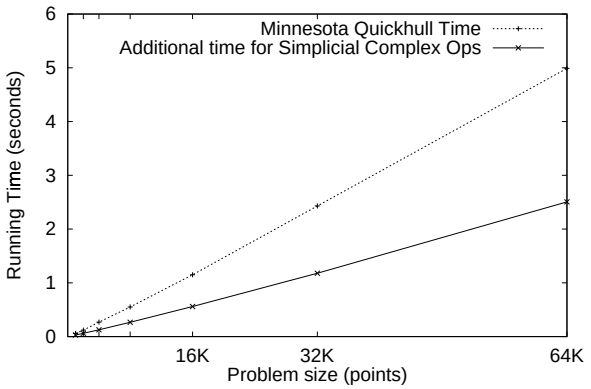


Figure 6: Running time as a function of the input size for both Minnesota Quickhull and for the C implementation of the dictionary operations. The distribution uses is OnSphere.

and the values given by the triangulation. Moreover, the triangulation so determined, when projected onto the plane, is required to have the Delaunay property[5] [4], as such triangulations have several desirable properties. However, since it is prohibitively expensive to compute a triangulation that is actually optimal, heuristics are typically employed that perform well in practice.

One such heuristic is the *greedy insertion heuristic.* The greedy insertion heuristic starts by dividing the plane into two triangles, and initializes a priority queue with one point from each triangle, the point having the greatest error between the sample value and the value given by the triangle. The heuristic then builds the triangulation incrementally, at each step obtaining the sample point with maximum error from the priority queue and updating the Delaunay triangulation to include that point.[6] The priority queue is then updated to include the points of maximum error for each new triangle. Typically only a few triangles are created in each step, resulting in only moderate rescanning of the terrain samples. This process is then repeated until an acceptable maximum error is achieved.

We implemented this heuristic using our persistent triangulation package. Delaunay triangulations can be computed using a three-dimensional convex hull procedure by projecting the points from the plane onto a paraboloid (the surface specified by the equation $z = x^2 + y^2$) and computing the convex hull of the projected points [4], so the implementation was straightforward. To measure its performance, we ran it on two sets of terrain sample data, one from the vicinity of Ozark, Missouri, and the other from the west end of Crater Lake, Oregon. A 1000-point triangulation of each of these data sets is given in Figures 7 and 8. As in the previous section, we counted key-comparisons and floating-point operations for each run. The results appear in Figure 9 and show that the number of key comparisons is significantly smaller than the number of floating-point operations, especially for the smaller sizes.

[5]The Delaunay property specifies that no point lies within the circumcircle of any triangle of which it is not a vertex, except in certain degenerate circumstances.

[6]An alternative greedy heuristic, designed to avoid narrow triangles, is to add the circumcenter of the triangle containing the point of maximum error, rather than the point of maximum error itself.

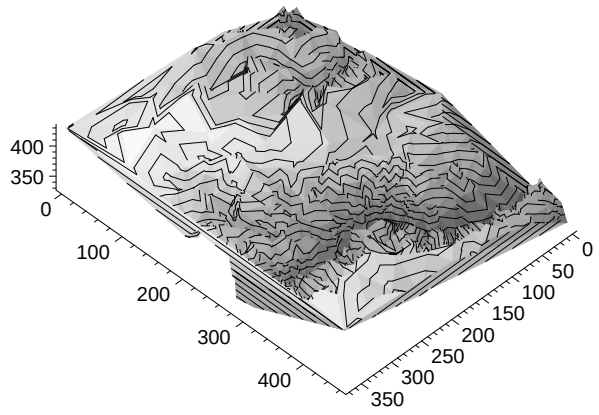


Figure 7: 1000-point triangulation of Ozark

6 Conclusion

Purely functional, persistent data structures offer a number of programming advantages over their more familiar, ephemeral counterparts. Many algorithms in computational geometry are based on low-level graph structures. The analysis of these algorithms is based on a unit-cost assumption for the fundamental operations on a graph. A naïve translation of these algorithms using a persistent tree structure to represent the graph would introduce an $O(\lg n)$ factor into the asymptotic complexity, resulting in asymptotically sub-optimal performance in the persistent case.

This paper addresses the question of whether this logarithmic penalty is avoidable in specific cases. We consider the fundamental problem of constructing the convex hull of a set of points in three dimensions. We give a high-level description of the hull as a simplicial complex, and provide a persistent implementation of it. We also present a new algorithm, called the bulldozer algorithm, that achieves the asymptotically optimal $O(n \lg n)$ time bound (in a randomized sense) that works with this abstract representation of the hull. To assess the practicality of the algorithm, we implemented this algorithm in Standard ML and measured its performance on a variety of artificial data sets. We also used this algorithm to build a terrain modelling application derived from work of Garland and Heckbert [13]. Our results confirm that the persistent representation of the hull, together with our new algorithm for constructing it, are both theoretically and practically efficient.

Acknowledgements

We thank Chris Okasaki for his red-black tree library, which we used in our implementation of simplices.

References

[1] Paul Alexandroff. *Elementary Concepts of Topology*. Dover Publications, Inc, New York, 1961.

[2] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, December 1996.

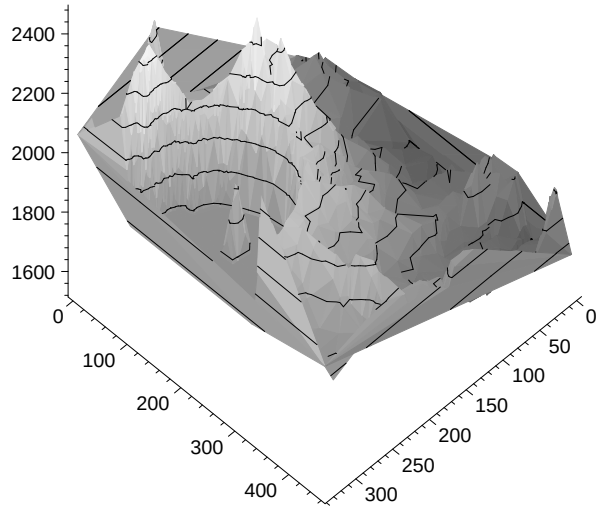


Figure 8: 1000-point triangulation of Crater Lake

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[5] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[6] Hal Burch and Gary Miller. Computing the convex hull in a functional language. In Preparation, September 1999.

[7] Bernard Chazelle. An optimal convex hull algorithm and new results on cuttings. In *FOCS*, pages 29–38, 1991.

[8] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry. *Discrete and Computational Geometry*, 4:387–421, 1989.

[9] Paul F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer-Verlag, August 1989.

[10] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[11] James R. Driscoll, Daniel D. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.

[12] Martin Erwig. Functional programming with graphs. In *Proc. ACM Sigplan International Conference on Functional Programming*, pages 52–55, June 1997.

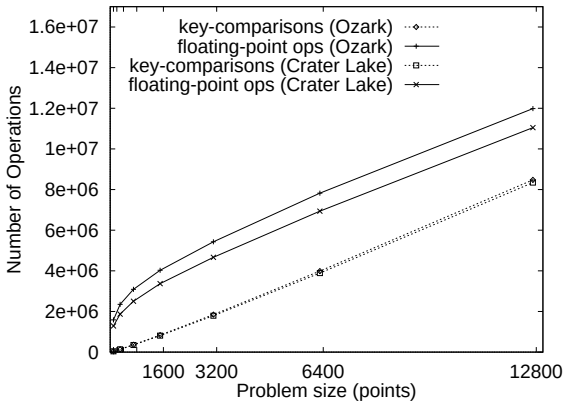


Figure 9: Operation counts as a function of input size.

[13] Michael Garland and Paul Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, CS Dept, Carnegie Mellon U., September 1995.

[14] P. J. Giblin. *Graphs, Surfaces, and Homology*. Chapman and Hall, London, 1977.

[15] C. M. Gold and P. R. Remmele. Voronoi methods in GIS. In *Algorithmic foundations of geographic information systems*, pages 21–35. Springer-Verlag, 1997.

[16] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[17] N. Gupta and S. Sen. An improved output-size sensitive parallel algorithm for hidden-surface removal for terrains. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 215–219, March 1998.

[18] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *ACM Symposium on Theory of Computing*, pages 93–102, May 1995.

[19] Rajeev Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995.

[21] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *CACM*, 29(7):669–679, 1986.

An Algebraic Dynamic Programming Approach to the Analysis of Recombinant DNA Sequences

Robert Giegerich[*] *Stefan Kurtz*[†] *Georg F. Weiller*[‡]

1 Introduction

1.1 From Biosequences to Structure to Function

Dynamic programming (DP, for short) is a fundamental programming technique, applicable to great advantage whenever the input to a problem spawns an exponential search space in a structurally recursive fashion, and solutions to subproblems adhere to an optimality principle. No wonder that DP is the predominant paradigm in computational (molecular) biology. Sequence data—DNA, RNA, and proteins—are determined on an industrial scale today. The desire to give a meaning to these molecular data gives rise to an ever increasing number of sequence analysis tasks. Given the mass of these data and the length of these sequences ($3 \cdot 10^6$ bases for a bacterial genome, $3 \cdot 10^9$ for the human genome), program efficiency is crucial. DP is used for assembling DNA sequence data from the fragments that are delivered by the automated sequencing machines [1], and to determine the intron/exon structure of genes [3]. It is used to infer function of proteins by homology to other proteins with known function [10, 11], and to determine the secondary structure of functional RNA genes or regulatory elements [15]. In some areas, DP problems arise in such variety that a specific code generation system for implementing the typical DP recurrences has been developed [2]. This system, however, does not support the development or validation of these recurrences.

1.2 Outline of Algebraic Dynamic Programming

The systematic development of DP solutions for problems in computational biology has been recently addressed by Giegerich [4]. There, an algebraic approach to dynamic programming (ADP) was developed and applied to the problem of folding an RNA sequence into its secondary structure. Here we will adapt ADP to the problem of comparing *two* sequences in the edit distance model. ADP is based on the following principles:

1. The analysis problem at hand is conceptually split into a *structure recognition* and a *structure evaluation* phase. Recognized structures are represented by an algebraic datatype $\mathcal{S}$. Evaluation is specified in terms of a particular $\mathcal{S}$-algebra.

2. A subset of well-formed structures in $\mathcal{S}$ is distinguished by a *tree grammar*. We require that

[*] Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany, E-mail: robert@techfak.uni-bielefeld.de, partially supported by a grant from the Australian National University

[†] Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany, E-mail: kurtz@techfak.uni-bielefeld.de, partially supported by DFG-grant Ku 1257/1-1

[‡] Bioinformatics Laboratory, Research School of Biological Science, Australian National University, Canberra, ACT 0200, Australia, E-mail: weiller@rsbs.anu.edu.au

- structure recognition finds *all and only all* well-formed structures,
- structure recognition constructs each such structure exactly *once*,
- structure evaluation is performed only on well-formed structures.

3. By providing *parsers* for the terminal symbols and *parser combinators* for the alternative, applicative, and sequential operators of the tree grammar, the grammar turns into a recognizer for its language.

4. A recursive recognizer is turned into a *DP algorithm* by *tabulation*: Each (recursive) parser is substituted by a (recursively defined) table of results. This is achieved by an efficiency annotation that does not change the declarative meaning of the grammar.

5. An *abstract evaluator* is a recognizer written in terms of an abstract $\mathcal{S}$-algebra, applying an abstract choice function to each intermediate result. Instantiated with a concrete $\mathcal{S}$-algebra, it interleaves structure recognition and evaluation. The concrete evaluator so obtained runs in polynomial time and space, if the concrete evaluation algebra has a constant time and space bound with respect to each intermediate result.[1]

6. *DP recurrences*, suitable for implementation in any imperative language, can be derived from the specification by straightforward substitution and program simplification.

1.3 Why Functional Programming Matters

ADP is a program development method, and the resulting program can (and normally will) eventually be implemented in an imperative language. A functional language like *Haskell*, however, makes the approach much more practical, and even enjoyable. The ADP approach can be completely embedded in *Haskell*, allowing us to experiment with executable programs at all stages of development. A wide range of lazy functional programming techniques is used, the most essential being parser combinators [9], programming with unknowns, and lazy (though immutable) arrays.

The productivity of the approach results from the modularity (cf. [8]) we achieve by separating structure recognition from structure evaluation. This advantage only exists in the functional paradigm; it is sacrificed in the final step (see Section 1.2, Principle 6).

Although ADP is a program development method, and not an equivalence transformation on programs, it bears some resemblance to deforestation [13], particularly in the form of [5]. The essential speed-up from exponential to polynomial time complexity, however, is not achieved by deforestation, but by tabulation and the simultaneous introduction of a choice function that reduces the volume of the intermediate results.

2 Biosequence Comparison in the Edit Distance Model

2.1 Searching for the Signals of Recombination

Comparison of DNA or protein sequences is predominantly done in the edit distance model. Two or more sequences are rearranged by introducing gaps, in a way that best exhibits their (dis)similarities. The concrete way in which distance or similarity is measured is expressed by means of a scoring function for matches, mismatches, and gaps. The scoring function varies from application to application. Sequence similarity is taken as an indication of homology, and multiple alignments or pairwise distances so obtained are frequently fed into programs that try to reconstruct phylogenies, i.e., evolutionary relationships of genes or species.

[1] More precisely, all operations of the algebra may be allowed to have polynomial efficiency, but the choice function is critical and must have a constant bound on the size of its output.

DNA *recombination* is an important mechanism in molecular evolution. Genes that have evolved independently in different strains of a virus, for example, may recombine in a new strain. This adds the power of parallel processing to Darwinian evolution, which is otherwise based on trial and error (i.e., random mutation and selection). In the presence of recombinant DNA, practically all commonly used analysis programs go wrong. There is no longer a tree-like phylogeny, as different parts of a sequence stem from different ancestors. In such a case, the best we can hope for from a tree reconstruction program is to tell us that there is no clear support for either of several possible trees in the distance data.

But there is a difference between data which are just noisy, and data which carry a clear signal about recombination events. There are different ways to explicitly search for recombination signals. The *PhylPro* program [14] does so by monitoring patterns of change in the mutual similarities in a multiple sequence alignment. In this paper, we take a direct approach, applicable to pairwise sequence alignment.

Traditionally, insertions and deletions are seen as random events, independent of their sequence context. But this is not totally adequate: Insertions and deletions in DNA sequences often stem from recombination events. The molecular mechanisms of recombination may leave traces in the form of target site duplications of varying length. Similar repeats may be formed through replication slippage, the other cellular process responsible for indel formation. Current methods of sequence analysis ignore these signals.

2.2 Extending the Edit Distance Model

Let x and y be two DNA sequences of length m and n, respectively. The classical edit distance model considers the following edit operations:

- $R\binom{a}{b}$ denotes the *replacement* of nucleotide a in x by b in y. If $a = b$, this is called a *match*, otherwise a *proper replacement*.
- $I\binom{-}{u}$ denotes the *insertion* of a non-empty sequence u of nucleotides into y, thereby introducing in x a gap of the same length, i.e., a sequence of $|u|$ dashes.
- $D\binom{u}{-}$ denotes the *deletion* of a non-empty sequence u of nucleotides from x, thereby introducing in y a gap of the same length, i.e., a sequence of $|u|$ dashes.

As new edit operations, we introduce recombinant deletion and insertion. Let t be a non-empty (but typically short) sequence of nucleotides that occurs both in x and in y.

- $S\left(\begin{smallmatrix} t & - & - \\ t & u & t \end{smallmatrix}\right)$ denotes a *recombinant insertion* in y: Following the *target site* t, present in both x and y, a sequence u of nucleotides is inserted into y, followed by a new copy of t in y. In x, a gap of the combined length of u and t is introduced.
- $L\left(\begin{smallmatrix} t & u & t \\ t & - & - \end{smallmatrix}\right)$ denotes a *recombinant deletion* from x: Following the *target site* t, present in both x and y, a sequence u of nucleotides is deleted from x. This requires a second copy of t to follow u in x. In y, a gap of the combined length of u and t is introduced.

In both cases, we allow the deleted or inserted sequence u to be empty, which makes the target site and its duplication form a tandem repeat in x or y.

Example 1 Here is an alignment of $attcgaa$ and $acgtatacgac$:

$$R\binom{a}{a}\ D\left(\begin{smallmatrix} t & t \\ - & - \end{smallmatrix}\right)\ S\left(\begin{smallmatrix} c & g & - & - & - & - & - & - \\ c & g & t & a & t & a & c & g \end{smallmatrix}\right)\ R\binom{a}{a}\ R\binom{a}{c}$$

It shows three replacements, a short deletion, and a recombinant insertion.

Proceeding from this operational view of recombination events to the analytic view, we must define the sequence pattern that can be interpreted in retrospect as a signal left from a recombination.

For any sequence z and any $i \in [0, |z|]$, $i{\downarrow}z$ denotes the suffix of z after dropping i symbols from the beginning of z. A *target site duplication in* y is a pair (i, j) such that $i{\downarrow}x = tz$ and $j{\downarrow}y = tutw$ for some t, u, w, z such that t is not empty. It is *maximal* if the first character of u, w, z is not the same, whenever these strings are not empty. A *target site duplication in* x is a pair (i, j) such that $i{\downarrow}x = tutw$ and $j{\downarrow}y = tz$ for some t, u, w, z such that t is not empty. It is *maximal* if the first character of u, w, z is not the same, whenever these strings are not empty.

Note that several target site duplications may be identified at the same position, differing in the length of the target sequence t. However, in the following we restrict to maximal duplications, since a longer target site duplication is to be taken as the stronger signal of a recombination event. We say that a recombinant deletion is *signalled* by a maximal target site duplication in x, and a recombinant insertion is *signalled* by a maximal target site duplication in y.

3 Computing Optimal Alignments in the Extended Edit Distance Model

3.1 An Algebraic Data Type for Extended Alignments

An alignment of x and y is traditionally represented by placing the aligned sequences on different lines, with inserted dashes to denote gaps. Successive dashes inside x are interpreted as an insertion into y, and successive dashes inside y as a deletion from x. The eye of the reader implicitly groups successive dashes into gaps of maximal length. A slightly more explicit view defines the alignment as a sequence of edit operations, with the additional restriction that a deletion (resp. insertion) must not immediately follow another deletion (resp. insertion).

With the new edit operations introduced here, we must resort to an even more explicit notation, marking target sites and their duplications. We also have to distinguish between gaps resulting from recombinations and gaps for which such an event is not indicated. We give up the view of a sequence of edit operations in favor of a recursive datatype `Alignment` with a constructor for each edit operation.

```
type Sequence a = Array Int a  -- indexed from 1
type Region = (Int,Int)        -- region (i,j) of x denotes x_{i+1} ... x_j
data Alignment a = R a (Alignment a) a |
                   D Region (Alignment a) |
                   I (Alignment a) Region |
                   S Region (Alignment a) Region Region Region |
                   L Region Region Region (Alignment a) Region |
                   Empty
```

Within the datatype `Alignment`, a target site duplication $i{\downarrow}x = tz$, $j{\downarrow}y = tutw$, is represented by an expression of the form `S t azw t u t`, wherein `azw` denotes an alignment of the suffixes z and w, and the three occurrences of `t` denote the target site in x and (duplicated) in y. Subwords of x and y are represented by their boundaries. Hence each edit operation requires constant space. If $k = |t|$ and $r = |u|$, then the above expression is actually written as

```
S (i,i+k) azw (j+k+r,j+k+r+k) (j+k,j+k+r) (j,j+k).
```

More space efficient representations are possible, since we only need to store i, j, k, and r.

Example 2 Given a datatype `Base` with constants `A`, `C`, `G`, and `T`, the expression

```
R A (D (1,3) (S (3,5) (R A (R A Empty C) A) (7,9) (3,7) (1,3))) A
```

denotes the alignment of $attcgaa$ and $acgtatacgac$ shown in Example 1. It may be printed in ASCII as

```
x = a t t c g - - - - - - - a a
y = a - - c g t a t a c g a c
    R D D S S U U U U T T R R
```

The third line indicates the edit operation involved. The recombinant insertion is labeled in the form `S U T` to indicate the starting target site `S`, the insert `U`, and the duplicated target site `T`.

At this point the reader is encouraged to take a look ahead at Section 4. It shows the improvement of standard alignment algorithms which we go for in the subsequent sections.

3.2 A Grammar for Well-Formed Alignments

The datatype `Alignment` is not specific enough to describe exactly all meaningful alignments. For example, it allows to represent two subsequent insertions, which should rather be merged into a single, longer insertion:

```
x = a t t c g - - - - - - - - - a a          -- malformed
y = a - - c g t a t a c g g g a c
    R D D S S U U U U T T I I R R
```

We do not accept a non-recombinant insertion immediately following a recombinant insertion. (We do, however, accept the opposite order.) It seems accidental to locate a duplicated target site in the middle of a gap. In such a situation, the alignment should rather show a single (non-recombinant) insertion (left alignment). Alternatively we might call for a recombinant insertion with a shorter target site (right alignment).

```
x = a t t c g - - - - - - - - - a a      x = a t t c g - - - - - - - - - a a
y = a - - c g t a t a c g g g a c        y = a - - c g t a t a c g g g a c
    R D D R R I I I I I I I I R R            R D D R S U U U U U U U T R R
```

It will be the task of the scoring function to choose between the latter two alternatives, while the malformed alignment above will not even be scored.

We introduce a grammar generating exactly the well-formed alignments. Following the discipline of [4], we use a tree grammar over the datatype `Alignment`, see Figure 1. The terminal symbols of this grammar are *base*, *region*, *uregion*, denoting a single nucleotide, a non-empty and an arbitrary sequence of nucleotides, respectively. The nonterminals are *alignment*, *noDel*, *noIns*, and *match*.

A production in this notation should be read as: "An alignment is either a *match*, or alternatively a *deletion* of some region from x followed by a *noDel*, or alternatively an *insertion* of some *region* in y, followed by a *noIns*."

As easily seen in the grammar *noDel* generates all alignments that do not start with a deletion. The use of *noDel* in the first production prevents successive deletions. Similarly for *noIns*. Leaving out the clauses for recombinant deletions and insertions, this tree grammar expresses the classical edit distance model [10, 11], used in biosequence analysis as well as in string processing.

The grammar still lacks some syntactic restriction: The three occurrences of *region* in the productions associated with `S` and `L` must all derive the same nucleotide sequence.

We now turn the grammar into a recognizer by defining terminal parsers and parser combinators [9]. For simplicity (and reasons of space) we assume that the input sequences x and y, as well as their length m and n are globally known. We do not show how these values are threaded through the functions.

A parser is given a pair of indices (i, j) and returns a list of all well-formed alignments of the suffix $i{\downarrow}x$ with the suffix $j{\downarrow}y$. Parsers for terminal symbols, however, are applied to one of the input sequences, so in their case, a call for (i, j) recognizes the subword (i, j) in either x or y. There is a parser combinator for

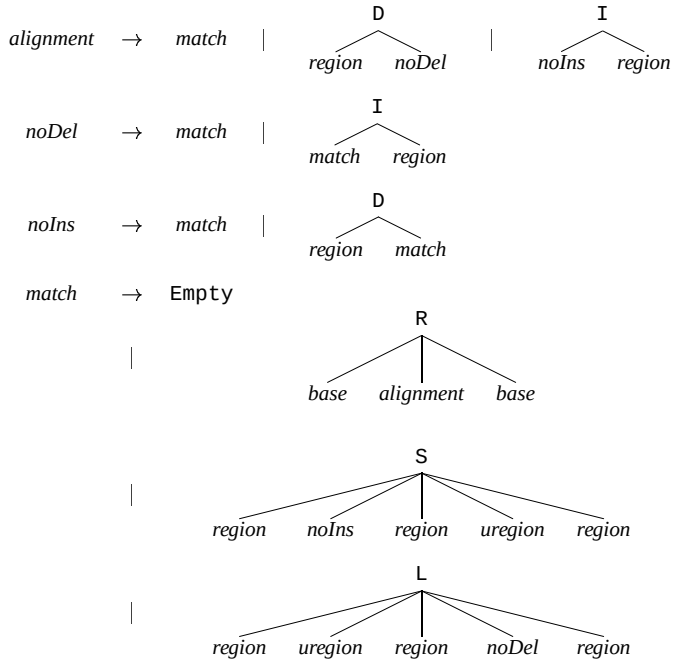


Figure 1: A tree grammar for well-formed alignments

the alternative, and for using parser results. Since we have two strings to process, there are two sequential combinators `+~~` and `~~+`. The combinators `-~~` and `~~-` are special forms of these.[2]

```
type Parser b = (Int,Int)->[b]              -- all parses of suffix pair

xbase,ybase::Parser a
xbase (i,j) = [x!j | i+1 == j]              -- recognize a base from x
ybase (i,j) = [y!j | i+1 == j]              -- recognize a base from y

region,uregion::Parser (Int,Int)
region  (i,j) = [(i,j) | i < j]             -- recognize a non-empty region
uregion (i,j) = [(i,j) | i <= j]            -- recognize any region

empty::b->(Parser b)
empty v (i,j) = [v | i == m && j == n]      -- recognize empty alignment

(|||)::(Parser b)->(Parser b)->(Parser b)
(|||) q r inp = q inp ++ r inp              -- alternative

(<<<)::(b->c)->(Parser b)->(Parser c)
(<<<) f q = map f.q                         -- using parser results

(+~~),(~~+),(-~~),(~~-)::(Parser (b->c))->(Parser b)->(Parser c)
(+~~) q r (i,j) = [s t | k<-[i..m], s<-q (i,k),   t<-r (k,j)  ]
(~~+) q r (i,j) = [s t | k<-[j..n], s<-q (i,k),   t<-r (j,k)  ]
(-~~) q r (i,j) = [s t | i < m,     s<-q (i,i+1), t<-r (i+1,j)]
(~~-) q r (i,j) = [s t | j < n,     s<-q (i,j+1), t<-r (j,j+1)]
```

[2] To explain our ideas it would sometimes suffice to present less *Haskell*-code. However, we want to make our paper self-contained and therefore show the code almost completely.

```
suchthat::(Parser b)->(b->Bool)->(Parser b)
suchthat q f inp = [s | s <- q inp, f s] -- check property of parser results

axiom::((Int,Int)->b)->b
axiom q = q (0,0)                        -- declare start symbol of grammar
```

In the grammar, written as a recognizer, we add syntactic restrictions for maximal target site duplication to the corresponding productions.

```
enum_alignments::(Eq a)=>(Sequence a)->(Sequence a)->[Alignment a]
enum_alignments x y = axiom alignment where

  alignment = match                                |||
              D <<< region +~~ noDel               |||
              I <<< noIns  ~~+ region

  noDel = match                                    |||
          I <<< match ~~+ region

  noIns = match                                    |||
          D <<< region +~~ match

  match = empty Empty                              |||
          R <<< xbase -~~ alignment ~~- ybase      |||
          recombIns                                |||
          recombDel

  recombIns = ((S <<< region +~~ noIns ~~+ region ~~+ uregion ~~+ region)
              `suchthat` targetsiteduplication)
              `suchthat` maximality

  recombDel = ((L <<< region +~~ uregion +~~ region +~~ noDel ~~+ region)
              `suchthat` targetsiteduplication)
              `suchthat` maximality
```

3.3 Dynamic Programming = Parsing + Tabulation

The above recognizer is easy to develop, but its associated parser is highly inefficient: Not only is there an exponential number of well-formed alignments for each pair of input sequences. The recognizer will also repeatedly parse the same subwords when called from different contexts.

The latter inefficiency is removed by introducing tabulation of intermediate parser results (representing alignments of suffixes of the two inputs). In other words, we employ DP. In contrast to memoization [7], DP uses explicitly and statically allocated tables.

```
type Parsetable b = Array (Int,Int) [b]

tabulated::Parser b->Parsetable b
tabulated q = array ((0,0),(m,n)) [((i,j),q (i,j)) | i<-[0..m],j<-[0..n]]
```

We modify the previous grammar, such that all parsers that do a non-constant amount of work per call shall use tabulation. Calling a parser means a table lookup. For reasons of space we only show the parser `alignment`. Note that our "efficiency annotation" does not affect the declarative meaning of the grammar.

```
dp_alignments::(Eq a)=>(Sequence a)->(Sequence a)->[Alignment a]
dp_alignments x y = axiom (alignment!) where

  alignment = tabulated (
              (match!)                  |||
              D <<< region +~~ (noDel!) |||
              I <<< (noIns!)  ~~+ region)
```

It is folklore knowledge that DP combines recursion and tabulation. After all, DP is normally formulated via matrix recurrences. The remarkable point here is the swiftness of transition, merely by adding the "keyword" `tabulated` and a few "`!`" to the grammar. The declarative and the operational meaning of the grammar remain unaffected, while efficiency improves from exponential to polynomial. If we had not been in love with *Haskell* before, this is where it would have happened.

The recognizer specified by this grammar runs in $O(n^2)$ space and in $O(n^6)$ time, due to the four sequential combinators in the productions associated with recombinant insertions and deletions.[3]

3.4 An $O(n^3)$ Implementation Using a Precomputed Lookahead

The above parser independently chooses three regions for the target site in x, in y, and for the duplication site in either x or y. Thereafter, those are checked for identity. Its efficiency can be greatly improved by the following observation: Consider a maximal target site duplication $i{\downarrow}x = tz$, $j{\downarrow}y = tutw$. Assume we have chosen and fixed the *combined* length $h = |tu|$ of the target site t and the insert u. Now for given x and y, there is really no variation left for the remaining constituents of the pattern:

- The start positions of the identical subwords must be i, j, and $j + h$.
- Their lengths are uniquely determined by the maximality condition.

Thus we will modify the parser to guess the position $j+h$, and then use a precomputed table `lookahead` to determine the length of t. For each $(i, j) \in [0, m] \times [0, n]$ this table stores the length of the longest common prefix of $i{\downarrow}x$ and $j{\downarrow}y$. It is computed and stored in $O(n^2)$ time and space. The overall running time of the recognizer is reduced to $O(n^3)$, while the space requirement remains $O(n^2)$. Note that since the three sites are now chosen as identical subwords of maximal length, this approach obviates the a-posteriori check for these properties. The resulting grammar is very similar to the grammar `ab_alignments` given in Section 3.5.

3.5 The Abstract Evaluator and Evaluation Algebras

According to [4], an abstract evaluator is obtained by abstracting from the constructors of the underlying datatype `Alignment`. Additionally, an abstract choice function is associated with each production, by the combinator `(...)`. Such an ensemble of functions of appropriate types constitutes an alignment-algebra.

```
type Algebra a b
  = (b,                                                    -- Empty
     a->b->a->b,                                           -- R
     (Int,Int)->b->b,                                      -- D
     b->(Int,Int)->b,                                      -- I
     (Int,Int)->(Int,Int)->(Int,Int)->b->(Int,Int)->b,     -- L
     (Int,Int)->b->(Int,Int)->(Int,Int)->(Int,Int)->b,     -- S
     [b]->[b])                                             -- choice function

(...)::Parser b->([b]->c)->(Int,Int)->c
(...) q choice = choice.q                -- applying a choice function
```

The abstract evaluator takes an alignment algebra as an additional parameter and adds the choice function.

[3] We generally assume that $m \in O(n)$, to simplify asymptotic results.

```
ab_alignments::(Eq a)=>(Algebra a b)->(Sequence a)->(Sequence a)->[b]
ab_alignments alg x y = axiom (alignment!) where

  (fE, fR, fD, fI, fL, fS, choice) = alg

  alignment = tabulated (
              (match!)                                        |||
              fD <<< region +~~ (noDel!)                      |||
              fI <<< (noIns!)  ~~+ region ... choice)

  noDel = tabulated (
          (match!)                                            |||
          fI <<< (match!) ~~+ region ... choice)

  noIns = tabulated (
          (match!)                                            |||
          fD <<< region +~~ (match!) ... choice)

  match = tabulated (
          empty fE                                            |||
          fR <<< xbase -~~ (alignment!) ~~- ybase             |||
          (recombIns!)                                        |||
          (recombDel!) ... choice)

  recombIns = tabulated (r ... choice) where
    r (i,j) = [fS t' noins d u t | l <-[j+1..n-1],
                                   let k = min h (lookahead!(i,l)),
                                   t'<- region (i,i+k),
                                   noins <- noIns!(i+k,l+k),
                                   d <- region (l,l+k),
                                   u <- uregion (j+k,l),
                                   t <- region (j,j+k)]
               where h = lookahead!(i,j)

  recombDel = tabulated (r ... choice) where
    r (i,j) = [fL t u d nodel t' | l <-[i+1..m-1],
                                   let k = min h (lookahead!(l,j)),
                                   t <- region (i,i+k),
                                   u <- uregion (i+k,l),
                                   d <- region (l,l+k),
                                   nodel <- noDel!(l+k,j+k),
                                   t'<- region (j,j+k)]
               where h = lookahead!(i,j)
```

The virtue of the abstract evaluator is, of course, that it can be called with arbitrary `Alignment` algebras: The *enumeration algebra* is trivially given by the constructors of the `Alignment` datatype.

```
enum_alg::Algebra a (Alignment a)
enum_alg = (Empty, R, D, I, L, S, id)
```

The *counting algebra* may be used to determine the number of well-formed alignments (without calculating the alignments, of course).

```
count_alg::Algebra a Int
count_alg = (fE, fR, fD, fI, fL, fS, choice) where
  fE       = 1
  fR _ x _ = x
  fD _ x   = x
  fI x _   = x
  fL _ _ _ x _ = x
  fS _ x _ _ _ = x
  choice [] = []
  choice xs = [sum xs]
```

The following scoring algebra implements a model with *affine* gap scores [6]. Such a model is used e.g. by *CLUSTALW*, a popular sequence alignment tool [12]. We have extended this algebra by scores for recombinant insertions and deletions. We have given a clear advantage to recombinant indels over regular ones by dividing their penalties by the length of the observed target site duplication.

```
affine_alg::Algebra Base Float
affine_alg = (fE, fR, fD, fI, fL, fS, choice) where
  fE   = 0
  fR a x b = x + matchscore a b
  fD (i,j) x = x + open + fromInt(j-i)*extend
  fI x (i,j) = x + open + fromInt(j-i)*extend
  fL (i,j) (u,u') _ x _ = x + ropen (i,j) + fromInt(u'-u)*rextend
  fS (i,j) x _ (u,u') _ = x + ropen (i,j) + fromInt(u'-u)*rextend
  choice [] = []
  choice xs = [minimum xs]
  open = 5.0
  extend = 0.2
  ropen (i,j) = open/fromInt(j-i)
  rextend = extend

matchscore::Base->Base->Float
matchscore a b | a == b    = 0
               | a > b     = matchscore' b a     -- function is symmetric
               | otherwise = matchscore' a b
                 where matchscore' A G = 1
                       matchscore' A _ = 3
                       matchscore' C G = 3
                       matchscore' C T = 1
                       matchscore' G T = 3
```

The *optimal alignment algebra* combines the scoring algebra with the enumeration algebra. This is straightforward. It returns an optimal alignment together with its score, in $O(n^2)$ space and $O(n^3)$ time.

4 Applications

We have applied our programs to chicken immunoglobin sequences taken from a multiple alignment. The typical improvements achieved by our algorithm are shown in Figure 2:

- In the left part of the recombinant alignment, a gap of length 12 (present in the multiple alignment) is re-discovered in the correct position. Additionally, it is marked as a direct repeat, as it may result from a recombinant insertion with an empty insert. Further experiments reveal that an alignment insensitive to recombination, but with the same scoring otherwise, has an insertion in approximately the same position, but does not exhibit the repeat due to an accidental ambiguity which causes one base to shift from the end to the beginning of the insert.

```
                                          >*    * ** * **     *   <
...tac------------tatggctggtaccag...ctccggttccctatccggctccacaggcacat...
...tactatggctggtactatggctggtaccag...ctccggctccccaggcagaaccacaagcacat...


                                          >                 *      <
...tactatggctggtac------------cag...ctccggttccctatccggctccacaggca------------cat...
...tactatggctggtactatggctggtaccag...ctccgg------------ctccccaggcagaaccacaagcacat...
...RRRSSSSSSSSSSSSTTTTTTTTTTTTTRRR...RLLLLLUUUUUUUTTTTRRRRRRRRSSSUUUUUUUUTTTRRR...
```

Figure 2: Original alignment (top) and recombinant alignment (bottom)

- The right part of the multiple alignment is poor with 8 mismatches (marked by the symbol *) within a region of 23 bases (between the delimiters > and <). The recombinant alignment offers an alternative explanation. It exhibits both a recombinant deletion and an insertion, with significant target sites, reducing the mismatch count to 1 over the same region as in the top alignment.

From the *Haskell*-program, DP recurrences were derived (see Section 1.2, Principle 6). Their implementation in C by a student required three days of work, including debugging. The functional program helped to spot errors in the C program that might otherwise have gone unnoticed. First measurements show that the C-program runs faster than the compiled *Haskell*-program by a factor of 68, while using 2% of the space.[4]

5 Conclusion

ADP is a method for algorithm development. It can be applied beneficially merely with pencil and paper. Its embedding in *Haskell* adds the convenience to test ideas very early, i.e., on a very high level of abstraction. The benefits of the functional methods are manyfold:

1. *Haskell*'s infix operators are notational convenience which is essential in this context.
2. The combinator parsing technique allows to have a consistent declarative and operational meaning of the grammar.
3. The equivalence of arrays and functions gives us polynomial efficiency without intellectual complication.
4. Laziness frees us from explicitly programming the order of computation of table entries, which is a most error-prone task in strict setting. Our experience is summarized in the motto "No subscripts, no errors".
5. Algebraic data types and higher order functions allow to separate recognition phase and evaluation algebra. m grammars and n evaluation algebras combine to $m \cdot n$ different analyses. In biosequence analysis, which involves much experimental programming, this compositionality takes the logarithm of the programming effort required otherwise.

The implementation effort can be summarized as follows. Having applied ADP in a different context before, it took an afternoon to adapt the combinator definitions and arrive at the $O(n^6)$ algorithm. Different evaluation

[4] For example, when computing the alignment of Figure 2 (for sequences of length 200), the C-program takes 5 seconds using 1.08 megabytes of space, while the Haskell program takes 340 seconds using 50 megabytes of space. These results were obtain on a Pentium PII computer with 300 MHz and 128 MB RAM. We used the C-compiler *gcc* version 2.7.2.3, and the *Haskell*-compiler *ghc* version 4.04-1.

algebras were helpful to test the program. Coming up with the lookahead based implementation required some thinking, but again, its implementation and testing was a matter of hours.

Although the improved parsers are "hard-coded" rather than defined via combinators, they fit in the rest of the program without friction. The flexibility makes us believe that the ADP method has virtually unlimited potential for improving programming productivity in biosequence analysis.

Acknowledgement We thank Matthias Höchsmann for implementing the recurrences in C and performing the experiments. Dirk Evers carefully read previous versions of the paper.

References

[1] E.L. Anson and E.W. Myers. ReAligner: A Program for Refining DNA Sequence Multi-Alignments. In *Proceedings of the First Conference on Computational Molecular Biology*, pages 9–16. ACM-Press, 1997.

[2] E. Birney and R. Durbin. Dynamite: A Flexible Code Generation Language for Dynamic Programming Methods Used in Sequence Comparison. In *Proc. of ISMB'97*, pages 56–64, 1997.

[3] M. A. Gelfand, L. I. Podolsky, T.V. Astakhova, and M. A. Roytberg. Recognition of Genes in Human DNA Sequences. *J. Comp. Biol.*, **3**(2):223–234, 1996.

[4] R. Giegerich. A Declarative Approach to the Development of Dynamic Programming Algorithms, Applied to RNA-Folding. Report 98–02, Technische Fakultät, Universität Bielefeld, 1998. `ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/Report98-02.ps.gz`.

[5] A. Gill, J. Launchbury, and S. Peyton-Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, June 1993*. ACM Press, New York, NY, 1993.

[6] O. Gotoh. An Improved Algorithm for Matching Biological Sequences. *J. Mol. Biol.*, **162**:705–708, 1982.

[7] J. Hughes. Lazy Memo-Function. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 129–146. Lecture Notes in Computer Science **201**, Springer Verlag, 1985.

[8] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, **32**(2):98–107, 1989.

[9] G. Hutton. Higher Order Functions for Parsing. *J. Functional Programming*, **3**(2):323–343, 1992.

[10] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins. *J. Mol. Biol.*, **48**:443–453, 1970.

[11] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Biol.*, **147**:195–197, 1981.

[12] J.D. Thompson, D.G. Higgins, and T.J. Gibson. CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Position Specific Gap Penalties and Weight Matrix Choice. *Nucleic Acids Res.*, **22**:4673–4680, 1994.

[13] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, **73**:231–248, 1990.

[14] G.F. Weiller. Phylogenetic Profiles: A Graphical Method for Detecting Genetic Recombinations in Homologous Sequences. *Mol. Biol. Evol.*, **15**(3):326–335, 1998.

[15] M. Zuker. The Use of Dynamic Programming Algorithms in RNA Secondary Structure Prediction. In *Mathematical Methods for DNA Sequences, Waterman, M.S. (editor)*, pages 159–184. 1989.

Constructing Red-Black Trees

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract

This paper explores the structure of red-black trees by solving an apparently simple problem: given an ascending sequence of elements, construct, in linear time, a red-black tree that contains the elements in symmetric order. Several extreme red-black tree shapes are characterized: trees of minimum and maximum height, trees with a minimal and with a maximal proportion of red nodes. These characterizations are obtained by relating tree shapes to various number systems. In addition, connections to left-complete trees, AVL trees, and half-balanced trees are highlighted.

1 Introduction

Red-black trees are an elegant search-tree scheme that guarantees $O(\log n)$ worst-case running time of basic dynamic-set operations. Recently, C. Okasaki [10, 11] presented a beautiful functional implementation of red-black trees. In this paper we plunge deeper into the structure of red-black trees by solving an apparently simple problem: given an ascending sequence of elements, construct a red-black tree that contains the elements in symmetric order. Since the sequence is ordered, the construction should only take linear time.

There are at least two ways of approaching this problem. One can try to analyse and to improve the standard method, which works by repeatedly inserting elements into an empty initial tree. Or one can build upon well-known algorithms for constructing trees of minimum height [4, 5]. In the latter case one must solve the following related problem: given an arbitrary binary search-tree, is there a way of coloring the nodes such that a red-black tree emerges?

We follow both paths as each provides us with different insights into the structure of red-black trees. Along the way, we will encounter several extreme red-black tree shapes: trees of minimum and maximum height, trees with a minimal proportion of red nodes, and others with a maximal proportion. In addition, connections to left-complete trees [13], AVL trees [2], and half-balanced trees [12] are highlighted.

2 Functional red-black trees

Let us start with a brief review of C. Okasaki's functional red-black trees [10, 11]. A red-black tree is a binary tree whose nodes are colored either red or black.

$$\begin{aligned} &\textbf{data } \mathit{Color} &&= \mathit{R} \mid \mathit{B} \\ &\textbf{data } \mathit{RBTree}\ a &&= \mathit{E} \mid \mathit{N}\ \mathit{Color}\ (\mathit{RBTree}\ a)\ a\ (\mathit{RBTree}\ a) \end{aligned}$$

The balance conditions are best explained if we take a look at their historical roots. Red-black trees were developed by R. Bayer [3] under the name *symmetric binary B-trees*. This term indicates that red-black trees were originally designed as binary tree representations of *2-3-4 trees*. Recall that a 2-3-4 tree consists of 2-, 3- and 4-nodes (a 3-node, for instance, has 2 keys and 3 children) and satisfies the invariant that all leaves appear on the same level. The idea of red-black trees is to represent 3- and 4-nodes by small binary trees, which consist of a black root and one or two auxiliary red children. This explains the following two balance conditions.

Red condition: Each red node has a black parent.

Black condition: Each path from the root to an empty node contains exactly the same number of black nodes (this number is called the tree's *black height*).

Note that the red condition implies that the root of a red-black tree is black.

The algorithm for inserting an element into a red-black tree is nearly identical to the standard algorithm for unbalanced binary trees. The main difference is that the constructor for building nodes, N, is replaced by a *smart constructor* [1] that maintains the invariants.

$$
\begin{array}{lcl}
\mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{RBTree}\ a \to \mathit{RBTree}\ a \\
\mathit{insert}\ a\ t & = & \mathit{blacken}\ (\mathit{ins}\ t) \\
\quad \textbf{where}\ \mathit{ins}\ E & = & N\ R\ E\ a\ E \\
\qquad \mathit{ins}\ (N\ c\ l\ b\ r) & & \\
\qquad\quad \mid a < b & = & \mathit{bal}\ c\ (\mathit{ins}\ l)\ b\ r \\
\qquad\quad \mid a == b & = & N\ c\ l\ a\ r \\
\qquad\quad \mid a > b & = & \mathit{bal}\ c\ l\ b\ (\mathit{ins}\ r) \\
\mathit{blacken}\ (N\ \_\ l\ a\ r) & = & N\ B\ l\ a\ r
\end{array}
$$

Since a new node is colored red, only the red condition is possibly violated. The smart constructor *bal* detects and repairs such violations.

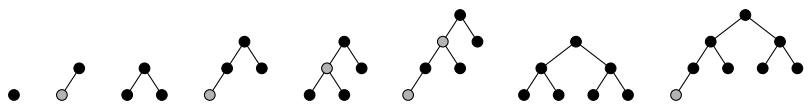
$$
\begin{array}{lcl}
\mathit{bal}\ B\ (N\ R\ (N\ R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\mathit{bal}\ B\ (N\ R\ t_1\ a_1\ (N\ R\ t_2\ a_2\ t_3))\ a_3\ t_4 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\mathit{bal}\ B\ t_1\ a_1\ (N\ R\ (N\ R\ t_2\ a_2\ t_3)\ a_3\ t_4) & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\mathit{bal}\ B\ t_1\ a_1\ (N\ R\ t_2\ a_2\ (N\ R\ t_3\ a_3\ t_4)) & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\mathit{bal}\ c\ l\ a\ r & = & N\ c\ l\ a\ r
\end{array}
$$

The simplest way to construct a red-black tree is to repeatedly insert elements into an empty initial tree.

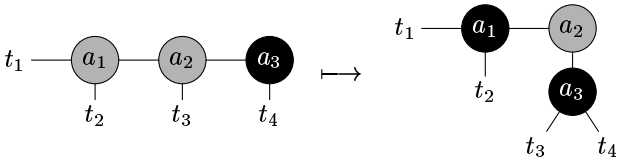
$$
\begin{array}{lcl}
\mathit{top\text{-}down} & :: & (\mathit{Ord}\ a) \Rightarrow [a] \to \mathit{RBTree}\ a \\
\mathit{top\text{-}down} & = & \mathit{foldr}\ \mathit{insert}\ E
\end{array}
$$

3 A closer look at *top-down*

What tree shapes does *top-down* produce when the given sequence is ascending? It is instructive to peek at some small examples first. The following trees are generated by *top-down* $[1\,..\,i]$ for $1 \leqslant i \leqslant 8$ ('$\circ$' is a red node and '$\bullet$' is a black node).

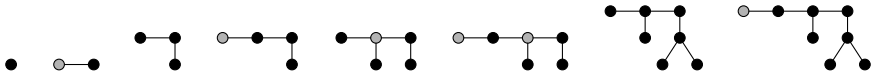


Note that we do not care to label the nodes as the keys are uniquely determined by the search-tree property. Since the list is processed from right to left, the elements are inserted in descending order. Consequently, *ins* always traverses the *left spine* of the tree to the leftmost leaf. The examples show that the color of the leftmost leaf alternates between black and red implying that the red condition is violated in every second step. Because *ins* always branches to the left, only the first equation of the smart constructor *bal* can possibly match. If we draw the left spine horizontally, the balancing operation takes the following form.



The balancing operation paints the first node black and combines the next two putting the black node below the second red one. Since this is the only operation applicable, all nodes not on the left spine must be black. We know even more: the black condition implies that the trees below the left spine (t_2, t_3 and t_4) must be perfectly balanced binary trees (perfect trees for short). Thus, the generated red-black trees correspond to sequences of *topped perfect trees*. A topped tree is a tree with an additional unary node on top. Topped perfect trees are, in fact, a widespread plant in the design and analysis of data structures. J.-R. Sack and T. Strothotte [13], who call them *pennants*, employ them to design algorithms for splitting and merging heaps in the form of *left-complete* binary trees. In a left-complete tree all leaves appear on at most two adjacent levels and the leaves on the lowest level are in the leftmost possible positions. We will exhibit further connections to their work in Section 6. The author recently showed that pennants also underly binomial heaps [7].

A topped perfect tree or a pennant of *rank* r is a perfect tree of height r with an additional node on top. It follows that a pennant of rank r contains exactly 2^r nodes. Turning to the analysis of *top-down* $[1\,..\,i]$ we are left with the task of determining the pennants' ranks. It is helpful to redraw our examples according to the *left-spine view*.



A pattern begins to emerge: let r be the rank of the rightmost pennant; the black condition implies that a pennant of rank *i* appears either once or twice for all $0 \leqslant i \leqslant r$. Since the size of a rank *i* pennant is 2^i, we have that the trees correspond to 'binary numbers' composed of the digits 1 and 2. It is worthwhile to study this number system, which we call 1-2 system, in more detail. Recall that the value of the radix-2 number $(b_{n-1} \ldots b_0)_2$ is $\sum_{i=0}^{n-1} b_i 2^i$. Since the number system abandons the digit 0 in favour of the digit 2, each natural number has, in fact, a unique representation. It is conceivable that the number system was already known in the middle ages when the number 0 was frowned upon and fell into oblivion later. Purists are probably attracted by the fact that there is no need to disallow leading zeros. Counting is easy:

$$()_2, (1)_2, (2)_2, (11)_2, (12)_2, (21)_2, (22)_2, (111)_2, (112)_2 \ldots$$

Note that 0 is represented by the empty sequence. The increment is similar to the ordinary binary increment; we have $s1 + 1 = s2$ and $s2 + 1 = (s + 1)1$. In the sequel we use the notation $(b_{n-1} \ldots b_0)_{1\text{-}2}$ to emphasize that the digits are drawn from the set $\{1, 2\}$.

We have seen that red-black trees generated by *top-down* $[1\,..\,n]$ are uniquely determined by the 1-2 decomposition of *n*. Let us examine some examples: $(1^{\{n\}})_{1\text{-}2} = 2^n - 1$ corresponds to a perfect tree of height n ($d^{\{n\}}$ means the digit d repeated n times); left-complete trees are produced for $(1^{\{n\}}2)_{1\text{-}2} = 2^{n+1}$ and $(21^{\{n\}}) = 3 \cdot 2^n - 1$. Hence, we know that *top-down* $[1\,..\,i]$ produces trees of minimum height for

some values of i. This is good news. On the other hand $(2^{\{n\}})_{1\text{-}2} = 2^{n+1} - 2$ and $(12^{\{n\}})_{1\text{-}2} = 3 \cdot 2^n - 2$ correspond to *skinny trees* of height $2n$ and $2n + 1$, respectively. A skinny tree is a tree of smallest possible size for a given height. Fig. 1 depicts a skinny tree of height 7 and its 'successor', which is a left-complete tree. Note that we cannot remove a single node from the skinny tree without either lowering the tree's height

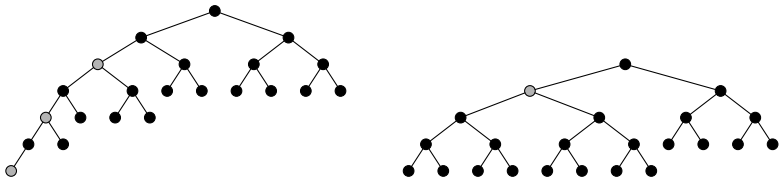


Figure 1: A skinny tree of height 7 and its 'successor'

or violating the black condition. Skinny trees give us a precise upper bound for the height of a red-black tree [14]:

$$\textit{height}\ t \leqslant 2\ \textit{lg}\ (\textit{size}\ t + 2) - 2\ .$$

So the bad news is that *top-down* $[1\,..\,i]$ produces red-black trees of maximum height for some values of i.

The trees generated by *top-down* have another intriguing property. They contain the minimal number of red nodes among all red-black trees of that size.

Sketch of proof. The central idea is to show that each tree of size i can be transformed into the shape generated by *top-down* $[1\,..\,i]$ and that the transformations do not increase the number of red nodes. We base the proof on 2-3-4 trees, which underly red-black trees, see Section 2. The shape of a 2-3-4 tree is uniquely represented by a sequence of level descriptions where each level description is a sequence of the numbers 2, 3 and 4. As an example, the trees of Fig. 1 are represented by

2	3
3 2	2 2 2
3 2 2 2 2	2 2 2 2 2 2
3 2 2 2 2 2 2 2 2 2 2 2	2 2 2 2 2 2 2 2 2 2 2 2 2 .

To simplify the proof we consider not only 2-3-4 trees but general multiway branching trees and transformations on these trees. To this end we generalize level descriptions to sequences of arbitrary natural numbers. We only require the following 'sanity' condition: A sequence of level descriptions $\ell_n \ell_{n-1} \ldots \ell_2 \ell_1$ is *valid* iff $|\ell_n| = 1$ and $\sum \ell_{i+1} = |\ell_i|$ for $1 \leqslant i < n$. We use two kinds of transformations. The first transformation replaces a subsequence of numbers in a certain level by another subsequence and is depicted $s \rightsquigarrow s'$. To ensure that the resulting tree is valid s and s' must satisfy $\sum s = \sum s'$ and $|s| = |s'|$. The second transformation is $s \overset{+k}{\rightsquigarrow} s'$ with $k \geqslant 1$, $\sum s = \sum s'$, and $|s| + k = |s'|$. Here, s is replaced by s' in a certain level and the level above is increased by k, ie one or more numbers in this level are increased by a total of k. If there is no level above, we silently create a new level consisting of a 1-node. The reader is invited to relate the two transformations to operations on multiway branching trees. Note that $s \rightsquigarrow s'$ does not affect the number of red nodes and that $s \overset{+k}{\rightsquigarrow} s'$ decreases the number of red nodes by k (recall that a node of size n consists of one black node and $n - 1$ red nodes). Now, an arbitrary 2-3-4 tree can be transformed into the desired shape by repeatedly applying the following transformations from the bottom level to the top level.

$$2\,3 \rightsquigarrow 3\,2 \qquad 3\,3 \overset{+1}{\rightsquigarrow} 2\,2\,2 \qquad (n+4) \overset{+1}{\rightsquigarrow} (n+2)\,2$$

In the resulting tree each level description has the form $[3]2^*$, ie an optional 3-node followed by an arbitrary number of 2-nodes. Since the tree generated by *top-down* satisfies the same property and since each number has a unique 1-2 decomposition, the claim follows. □

Using the 1-2 number system we can even quantify the minimal number of red nodes: a red-black tree of size n contains at least k red nodes where k is the number of 2's in the 1-2 representation of n.

4 Improving *top-down*

The analogy to the 1-2 number system can be exploited to give a better implementation of *top-down* for the special case that the elements appear in ascending order. The digits become containers for pennants:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Digit}\ a & = & \mathit{One}\ a\ (\mathit{RBTree}\ a) \\ & \mid & \mathit{Two}\ a\ (\mathit{RBTree}\ a)\ a\ (\mathit{RBTree}\ a)\ . \end{array}$$

A red-black tree is represented by a list of digits in increasing order of size (the least significant digit comes first). Inserting an element corresponds to incrementing a 1-2 number. The function *incr*, which does the job, essentially implements the two laws $s1+1 = s2$ and $s2+1 = (s+1)1$ where s is any sequence of 1-2 digits.

$$\begin{array}{lcl} \mathit{incr} & :: & \mathit{Digit}\ a \to [\mathit{Digit}\ a] \to [\mathit{Digit}\ a] \\ \mathit{incr}\ (\mathit{One}\ a\ t)\ [\,] & = & [\mathit{One}\ a\ t] \\ \mathit{incr}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{One}\ a_2\ t_2 : ps) & = & \mathit{Two}\ a_1\ t_1\ a_2\ t_2 : ps \\ \mathit{incr}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{Two}\ a_2\ t_2\ a_3\ t_3 : ps) & = & \mathit{One}\ a_1\ t_1 : \mathit{incr}\ (\mathit{One}\ a_2\ (N\ B\ t_2\ a_3\ t_3))\ ps \end{array}$$

The reader is invited to relate *incr* to the definitions of *ins* and *bal* given in Section 2. The rest is easy: we repeatedly insert elements into the list of digits; the final result is converted to a red-black tree.

$$\begin{array}{lcl} \mathit{bottom\text{-}up} & :: & [a] \to \mathit{RBTree}\ a \\ \mathit{bottom\text{-}up} & = & \mathit{linkAll} \cdot \mathit{foldr}\ \mathit{add}\ [\,] \\ \mathit{add}\ a\ ps & = & \mathit{incr}\ (\mathit{One}\ a\ E)\ ps \\ \mathit{linkAll} & = & \mathit{foldl}\ \mathit{link}\ E \\ \mathit{link}\ l\ (\mathit{One}\ a\ t) & = & N\ B\ l\ a\ t \\ \mathit{link}\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) & = & N\ B\ (N\ R\ l\ a_1\ t_1)\ a_2\ t_2 \end{array}$$

It is a routine matter to prove *bottom-up* correct. We must essentially show that *add* implements *insert* on the left-spine view (*labels* lists the labels of a red-black tree):

$$\mathit{all}\ (a<)\ (\mathit{labels}\ t) \implies \mathit{linkAll}\ (\mathit{add}\ a\ t) = \mathit{insert}\ a\ (\mathit{linkAll}\ t)\ .$$

The reimplementation of *top-down* is worth the effort: a standard amortization argument shows that *bottom-up* takes only linear time.

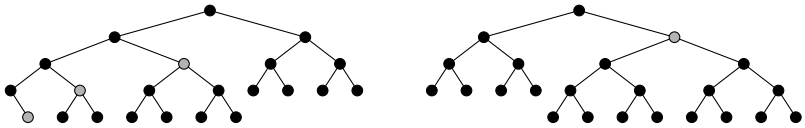
Remark. Red-black trees under the left-spine view correspond closely to *finger search-trees* [6]. A finger search-tree is a representation of an ordered list that allows for efficient insertion in the vicinity of certain points, termed fingers. Here we have a single static finger at the front end of the list. This data structure may be of further interest because it makes a nice implementation of updatable priority queues, which support deleting and decreasing a key.

5 Less height, please!

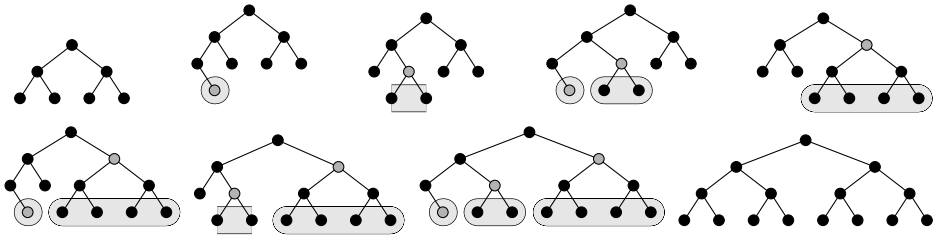
Having succeeded in implementing *top-down* efficiently, let us now try to reduce the height of the generated trees. It is well-known [8] that a binary search-tree is optimal if all leaves appear on at most two adjacent levels—under the assumption that all keys are equally likely. It turns out that it is almost trivial to modify *bottom-up* so that it produces trees of that shape. A simple *rotation to the right* suffices:

$$\mathit{link}'\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) \quad = \quad N\ B\ l\ a_1\ (N\ R\ t_1\ a_2\ t_2)\ .$$

Here are the shallow variants of the trees shown in Fig. 1.



It is interesting to see how the leaves on the bottom level are arranged. The pattern becomes apparent if we take a look at a longer sequence of trees.



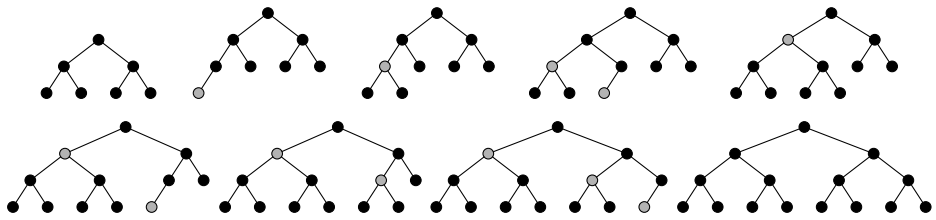
In each case the leaves appear on two adjacent levels. The definition of *link′* brings about that the leaves on the lowest level are descendants of a red node (apart from perfect trees). Depending on the position of the red node we have either a group of 1, 2 or 4 nodes as indicated by the shading. If we convert the groups into binary digits, the binary numbers $(000)_2$, $(001)_2$, $(010)_2$, …, $(111)_2$ appear on the last level. The number system helps to explain why this is the case. The 1-2 number $(b_{n-1}\ldots b_0)_{1\text{-}2}$ can be decomposed into two binary numbers: $(b_{n-1}\ldots b_0)_{1\text{-}2} = (1\ldots 1)_{0\text{-}1} + (b'_{n-1}\ldots b'_0)_{0\text{-}1}$ with $b'_i = b_i - 1$. The number $(1\ldots 1)_{0\text{-}1}$ corresponds to the perfect tree on top; the residue $(b'_{n-1}\ldots b'_0)_{0\text{-}1}$ to the leaves on the bottom level.

6 Digression: Left-complete binary heaps

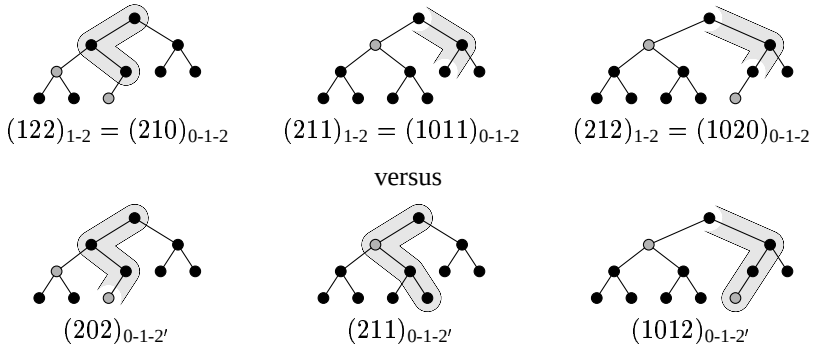
The generated trees, which we call *quasi left-complete trees*, are closely related to *left-complete trees*, which have the leaves on the lowest level in the leftmost possible positions. Consider the parents of the red nodes. If we *swap* their children, we obtain a left-complete tree. Again, it is trivial to modify *link* accordingly:

$$\mathit{link}''\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) \quad = \quad N\ B\ (N\ R\ t_1\ a_2\ t_2)\ a_1\ l\ .$$

To complete the picture here are the left-complete colleagues of the trees above.



Of course, the above transformation does not preserve the search-tree property. So let us assume in this section that we deal with *binary heaps* instead. It is not difficult to adopt *incr* to the new situation. We change our point of view because this provides an interesting link to the work of J.-R. Sack and T. Strothotte [13]. The decomposition of a left-complete tree into a list of pennants also lies at the heart of their algorithms for splitting and merging heaps. There is, however, a slight difference. They decompose a left-complete heap along the path from the root to the last leaf, ie the rightmost leaf on the last level. This seems to be an obvious choice but as we shall see gives rise to a more complicated number system. Here are the two ways of decomposing a left-complete tree.

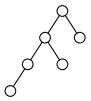


The difference is really minor: in the first row we follow the path to the first free position; in the second row to the last occupied position. Given a left-complete tree of size n, the first choice yields $\lfloor \lg(n+1) \rfloor$ pennants while the latter choice gives $\lceil \lg(n+1) \rceil$ pennants. Let us examine the number system corresponding to the latter choice, which we call 0-1-2′ system, for want of a better name. The examples show that the numbers are composed of the digits 0, 1 and 2. The digit d appears in the i-th position iff the path contains d pennants of size 2^i. For instance, the rightmost tree contains one pennant of size 8, one of size 2, and two of size 1. Consequently, the corresponding number is $(1012)_{0\text{-}1\text{-}2'}$. Without further restrictions the 0-1-2′ binary system is clearly *redundant*. It turns out that the number $(b_{n-1} \ldots b_0)_{0\text{-}1\text{-}2'}$ corresponds to a left-complete tree iff $m+1 \leqslant \sum_{i=0}^{m} d_i \leqslant m+2$ for all $m < n$ [13]. This condition implies that we never have two successive 2's. In fact, 21^*2 cannot appear as a subsequence (d^* means d repeated arbitrarily often). Incrementing a 0-1-2′ number is funny: first make a 'normal' increment (this can be done in constant time since the subsequence 22 is forbidden); then apply the transformation $21^{\{n\}}2 \mapsto 101^{\{n\}}2$ (at most once). If a segmented representation is used [10, Section 9.2.4], the latter transformation can also be done in constant time.

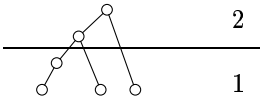
How do the number systems relate to each other? Well, we obtained the left-complete trees using rotations and swaps. A rotation corresponds to the carry propagation $2 \mapsto 10$ turning a 1-2 number into a 0-1-2 number. A swap does not affect the numeric representation. Thus, $(122)_{1\text{-}2}$ becomes $(210)_{0\text{-}1\text{-}2}$ and $(211)_{1\text{-}2}$ becomes $(1011)_{0\text{-}1\text{-}2}$. Note that the last digit of the 0-1-2 number is either 0 or 1. If we increment the last digit, we get the 0-1-2′ number corresponding to its successor. This relation is not too surprising since the path to the first free position corresponds to the path to the last element in the successor tree.

7 Coloring binary search-trees

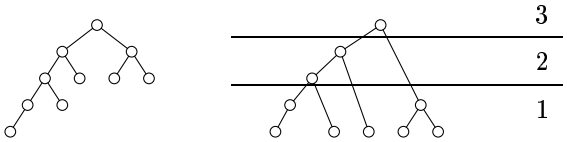
Let us now approach the problem of constructing red-black trees from a different perspective. Say, we are given an arbitrary binary search-tree and we are asked to color the nodes such that a red-black tree emerges or to report that it is not possible to do so. Here is the first test case.



The longest path from the root to an empty tree comprises 4 nodes; the shortest path consists of 2 nodes. Thus, the black height must be two and the nodes on the longest path must be colored black, red, black and red. We get a better picture of the situation if we draw the tree slightly different. In the picture below the vertical position of a node corresponds to its height rather than its distance from the root.



We additionally assign a *level number* to each node: a node of height h receives the level number $\lceil h/2 \rceil$. This way we divide the tree two-levelwise from the bottom to the top. Coloring is now easy: a node is colored red iff it has a parent with the same level number. Does this scheme work in general? Not quite, as the second test case shows.



All nodes of the right subtree must be colored black but our scheme colors the two leaves red. Fortunately, the picture on the right also contains an indication of the failure: an edge crosses two levels, ie the level numbers of two adjacent nodes differ by more than one. We can remedy this defect by lifting the right son of the root to the second level. Generally, it is possible to adjust the level numbers in a single top-down pass. It may, of course, happen that the leaves no longer appear on the same level. In this case the given tree is not colorable.

We are now ready to tackle the implementation. For simplicity, let us assume that the nodes of the input tree are decorated with the level number, ie trees are given as elements of the data type

$$\begin{array}{lcl} \textbf{type}\ \mathit{Level} & = & \mathit{Int} \\ \textbf{data}\ \mathit{Tree}\ a & = & \mathit{Empty} \mid \mathit{Node}\ \mathit{Level}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\ , \end{array}$$

in which $\lceil \mathit{height}\ (\mathit{Node}\ h\ l\ a\ r)\ /\ 2 \rceil = h$. The algorithm takes the following form:

$$\begin{array}{lcl} \mathit{rbtree} & :: & \mathit{Tree}\ a \to \mathit{RBTree}\ a \\ \mathit{rbtree}\ \mathit{Empty} & = & E \\ \mathit{rbtree}\ (\mathit{Node}\ h\ l\ a\ r) & = & N\ B\ (\mathit{rbtree}'\ h\ l)\ a\ (\mathit{rbtree}'\ h\ r) \\ \mathit{rbtree}' & :: & \mathit{Level} \to \mathit{Tree}\ a \to \mathit{RBTree}\ a \\ \mathit{rbtree}'\ hp\ \mathit{Empty} \mid hp == 1 & = & E \\ \qquad\qquad \mid \mathit{otherwise} & = & \mathit{error}\ \texttt{"not a red-black tree"} \\ \mathit{rbtree}'\ hp\ (\mathit{Node}\ h\ l\ a\ r) & = & N\ \mathit{color}\ (\mathit{rbtree}'\ h'\ l)\ a\ (\mathit{rbtree}'\ h'\ r) \\ \quad \textbf{where}\ h' & = & h\ `\mathit{max}`\ (hp-1) \\ \qquad \mathit{color} \mid hp == h & = & R \\ \qquad\qquad \mid \mathit{otherwise} & = & B\ . \end{array}$$

The auxiliary function *rbtree′* receives two arguments: the uncolored tree and the level number, *hp*, of the tree's parent. If the tree is empty, the level number must necessarily be one. Otherwise, the given tree

cannot be colored. The root of a non-empty tree is colored red iff its level number h coincides with hp. The adjusted level number of the root, which is passed to the recursive calls of *rbtree′*, is given by the expression h `max` $(hp - 1)$. Note that the level number of the input tree equals the black height of the generated tree. Furthermore, the longest path in the red-black tree contains alternating black and red nodes (if the height is odd and greater than one, the path starts with two black nodes). This in turn implies that *rbtree* produces trees of minimum black height.

It is relatively easy to see that *rbtree* yields a valid red-black tree. The converse is not so obvious: can we be sure that the given tree is not colorable if *rbtree* signals an error? It turns out that the correctness of the algorithm is best shown using an alternative characterization of red-black trees. Define the *min-height* of a tree as the length of the shortest path from the root to an empty node and the *max-height* as the length of the longest path. A binary tree t is said to be *half-balanced* [12] if for every subtree u of t,

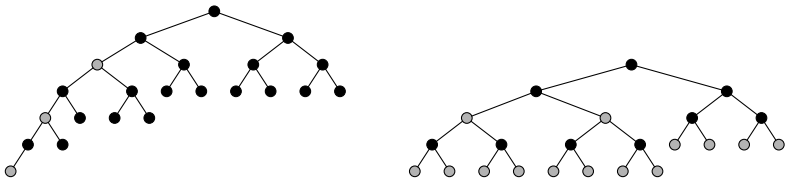
$$\tfrac{1}{2} \leqslant \textit{min-height}\ u\ /\ \textit{max-height}\ u \quad .$$

Every red-black tree is half-balanced because $\textit{height}\ t \leqslant 2\,\textit{black-height}$ and $\textit{black-height}\ t \leqslant \textit{min-height}\ t$. The function *rbtree* can be viewed as a constructive proof of the reverse implication. One must essentially show that the first parameter of *rbtree′* satisfies the following invariant

$$\tfrac{1}{2}\,(\textit{max-height}\ t + 1) \leqslant hp \leqslant \textit{min-height}\ t + 1 \quad .$$

If the input tree satisfies the AVL property [2], the algorithm can be slightly simplified: the test $hp == 1$ becomes obsolete and h' may be safely replaced by h. The resulting function already appears in the seminal paper on red-black trees [3, p. 295].

It is high time to see the algorithm in action. Here are the colored variants of the trees shown in Fig. 1.



It is not hard to show that *rbtree* produces trees with a maximal proportion of red nodes. However, we already know that the skinny tree on the left hand side contains the smallest possible number of red nodes (see Section 3). Both results imply that there is exactly one way of coloring skinny trees.

If *rbtree* is applied to a left-complete tree or to a tree generated by *bottom-up′*, it produces a red-black tree that contains the maximal possible number of red nodes among all trees of that size. Note that it is actually desirable that a tree contains many red nodes since the balancing operation *bal* takes only black nodes into account. To summarize: let *quasi-left-complete* be a variant of *bottom-up′* that constructs an uncolored tree of type *Tree a*. Then

$$\begin{array}{lcl} \textit{build} & :: & [a] \to \textit{RBTree}\ a \\ \textit{build} & = & \textit{rbtree} \cdot \textit{quasi-left-complete} \end{array}$$

builds a red-black tree that has minimum path length and a maximal portion of red nodes.

Sketch of proof. It remains to show that *build* constructs a red-black tree with a maximal portion of red nodes. The proof is largely analogous to the one in Section 3. This time we use transformations that do not decrease the number of red nodes. In particular, we employ the transformation $s \overset{-k}{\rightsquigarrow} s'$ with $k \geqslant 1$, $\sum s = \sum s'$, and $|s| - k = |s'|$. This transformation replaces s by s' in a certain level and decreases the level above by k (numbers must not become negative). A given 2-3-4 tree can be transformed into the desired shape by

repeatedly applying the following transformations from bottom to top (permutations such as $3\,2 \rightsquigarrow 2\,3$ are omitted from the list).

$$0 \overset{-1}{\rightsquigarrow} \epsilon \qquad 1\,1 \overset{-1}{\rightsquigarrow} 2 \qquad 1\,2 \overset{-1}{\rightsquigarrow} 3 \qquad 1\,3 \overset{-1}{\rightsquigarrow} 4 \qquad 1\,4 \rightsquigarrow 2\,3$$
$$2\,2 \overset{-1}{\rightsquigarrow} 4 \qquad 3\,3 \rightsquigarrow 2\,4$$

We tacitly assume that levels that consist only of a singleton 1-node are silently removed. In the resulting tree each level has the form $[2][3]4^*$, ie an optional 2-node followed by an optional 3-node followed by an arbitrary number of 4-nodes. Using an induction on the length of the left spine one can show that the trees generated by *build* can be transformed into the same shape using only '$\rightsquigarrow$' transformations. Finally, trees of this shape are uniquely determined by the size since they correspond to quaternary numbers composed of the digits 1, 2, 3 and 4 and since this number system is non-redundant. □

Remark. In solving the problem of constructing red-black trees we have answered quite a few exercises to be found in textbooks on data structures and algorithms, most notably exercises 10.9, 10.10 and 10.14 in [14] and exercises 3.9 and 9.7 in [10].

8 Acknowledgement

I would like to thank Chris Okasaki for his valuable comments on an earlier draft of this paper. Adam Buchsbaum, Gerth Stølting Brodal, and two anonymous referees made several helpful suggestions. Thanks are also due to Joachim Korittky for implementing *functional* METAPOST [9], which was used for drawing the diagrams.

References

[1] Stephen Adams. Functional Pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.

[2] G.M. Adel'son-Vel'skiĭ and Y.M. Landis. An algorithm for the organization of information. *Doklady Akademiia Nauk SSSR*, 146:263–266, 1962. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.

[3] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] Richard S. Bird. Functional algorithm design. *Science of Computer Programming*, 26:15–31, 1996.

[5] Richard S. Bird. Functional Pearl: On building trees with minimum height. *Journal of Functional Programming*, 7(4):441–445, July 1997.

[6] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 49–60, Boulder, Colorado, May 1977.

[7] Ralf Hinze. Functional Pearl: Explaining binomial heaps. *Journal of Functional Programming*, 9(1):93–104, January 1999.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, 2nd edition, 1998.

[9] Joachim Korittky. *Functional* METAPOST. Diplomarbeit, Universität Bonn, December 1998.

[10] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[11] Chris Okasaki. Functional Pearl: Red-Black trees in a functional setting. *Journal of Functional Programming*, 9(4), July 1999. To appear.

[12] H.J. Olivié. A new class of balanced search trees: Half-balanced search trees. *RAIRO Informatique théoretique*, 16:51–71, 1982.

[13] Jörg-Rüdiger Sack and Thomas Strothotte. A characterization of heaps and its applications. *Information and Computation*, 86(1):69–86, May 1990.

[14] Derick Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley Publishing Company, 1993.

An experimental study of compression methods for functional tries

Jukka-Pekka Iivonen
Nokia Telecommunications
iivonen@iki.fi

Stefan Nilsson
Royal Institute of Technology
snilsson@nada.kth.se

Matti Tikkanen
Nokia Telecommunications
matti.tikkanen@nokia.com

Abstract
We develop compression methods for functional tries and study them experimentally. Trie compression is usually implemented either as a combination of path compression and width compression or as a combination of path compression and level compression. We develop a new efficient implementation for width compression and show that in functional tries the combination of all of the three compression methods yields the best results when taking into account the trie size, the trie depth, the copy cost, and the search and update performance. We also show that the 2-3 tree minimizes the copy cost in balanced trees and compare our experimental results for tries to approximate analytical results for internal and external 2-3 trees. Our conclusion is that the path-, width-, and level-compressed trie is an ideal choice for a functional main-memory index structure.

Keywords
functional data structures, imperative data structures, trie, shadowing, path copying, path compression, width compression, level compression

1 Introduction

We conduct an experimental study of compression methods for functional tries. The reason why we choose to study trie structures using experimental rather than analytical methods is that the form of a trie depends on the kind of data stored. Analytic studies always need to use a data model, such as uniform distribution, independent random samples from a certain distribution function, or Bernoulli-type processes. It is well known that tries have a low average depth for all these types of data, but it is not immediately clear how these results translate to real world data. In this study we have used samples of English text, Internet routing tables, and geographic point data in addition to synthetic data.

We are currently designing and implementing a prototype-based strict functional database programming language called Shines. Shines will provide the programmer with built-in maps and sets that are implemented by persistent functional tries. In order to obtain maximal computational efficiency, we have implemented our functional data structures in C on top of Shades [23], a persistent functional heap supporting automatic disk-backed memory management in a soft real-time environment. Shades replaces logging with shadowing combined with real-time generational stop-and-copy garbage collection. Shadowing is essentially a synonym for the functional update policy where an update can be viewed as a three-step procedure. First a copy of the memory cell subjected to the update is created. Then the actual update is performed on the copy, and last a reference to the memory cell holding the copy is returned. In recursive structures such as trees an update leads to shadowing the entire search path. This is also known as path copying [26].

In its original form the trie [11, 12] is a data structure where a set of strings from an alphabet containing m characters is stored in an m-ary tree and each string corresponds to a unique path. There are several methods for implementing trie structures in the literature [4, 5, 7, 16, 19]. One of the drawbacks of these methods is that they need considerably more memory than a balanced search tree due to empty subtrees. We show how to avoid this problem by choosing a small alphabet and applying trie compression.

The average-case behavior of trie structures has been the subject of thorough theoretical analysis [10, 17, 24, 25]; an extensive list of references can be found in Handbook of Theoretical Computer Science [18]. The expected average depth of a trie containing n independent random strings from a distribution with density function $f \in L^2$ is $\Theta(\log n)$ [6]. This result holds also for data from a Bernoulli-type process [8, 9].

The best known compression technique for tries is path compression. The idea is simple: paths consisting of a sequence of single-child nodes are compressed, as shown in Figure 1b. A path compressed binary trie is often referred to as a Patricia tree. Path compression may reduce the size of the trie dramatically. In fact, the number of nodes in a path compressed binary trie storing n keys is $2n - 1$. The asymptotic expected average depth, however, is typically not reduced [15, 17].

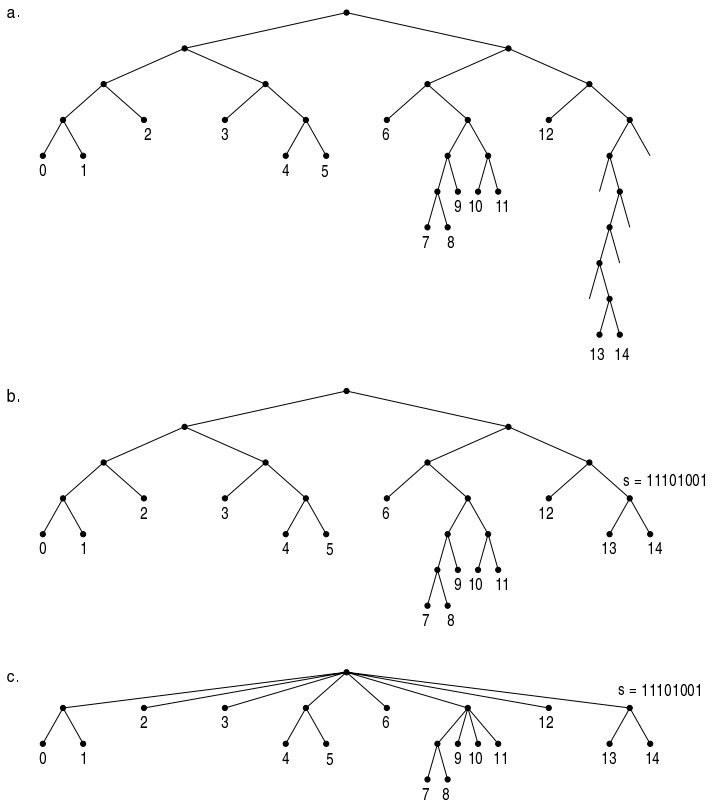


Figure 1: A binary trie (a), a path-compressed binary trie (b), and a path- and level-compressed binary trie (c). The data set consists of 15 string elements. Complete strings are stored in the leaves. The labels $0 - 14$ indicate the presence of an element, not a value stored. Variable s gives the longest common prefix string of a path-compressed node.

Level compression [1] is a more recent technique. Once again, the idea is simple: complete nodes (all children are non-empty) are compressed, and this compression is performed top down, see Figure 1c. The level-compressed trie, LC-trie, has proved to be of interest both in theory and practice. It is known that the average expected depth of an LC-trie is $O(\log \log n)$ for data from a large class of distributions [2]. This should be compared to the logarithmic depth of uncompressed and path compressed tries. These results also translate to good performance in practice, as shown by a recent software implementation of IP routing tables using an LC-trie [20].

The LC-trie was originally a static data structure that did not support inserts and deletes. To our knowledge, the LPC-trie [21] was the first dynamic variant of the LC-trie. The LPC-trie is an imperative data structure, however. In functional tries, an update leads to copying the entire search path from the root to the leaf subject to the update. Level-compressed nodes may have large out-degrees which makes functional updates extremely costly. To implement a functional LPC-trie, a possible approach to avoiding excessive copying in updates is to restrict the applicable out-degrees to 2, 4, 8, and possibly 16. The simple mathematical exercise below reveals, however, that a binary trie is not a good starting point if our goal is to keep the cost of copying as small as possible.

Let us assume that our trie is of a degree k and all leaves are at the same level in the trie. The approximate copy cost is given by the formula $f(k) = (k + a) \log_k N$ where a is the size of administrative data in a trie node, $k + a$ the size of the trie node in words, N the number of leaves (or elements stored), and $\log_k N$ the path length. With $a = 1$ the minimum of $f(k)$ is given by $f'(k) = 0$, i.e. $k \ln k - k = 1$. Computing the approximate solution for k yields $k = 3.59$. The discrete values closest to the minimum are 3 and 4, and $f(3) > f(4)$ for all $N > 1$. Thus, a functional quaternary trie minimizes the copy cost.

The rationale above holds only for tries storing uniformly distributed data or clustered data. For other distributions of data the topmost trie nodes cannot be assumed to have only few empty children. Empty subtries increase the cost of copying and the overall memory consumption. We have at our disposal yet another compression method, however, that can be used for removing the empty subtries: width compression. In a width-compressed trie, the out-degree of the trie is fixed, but only non-empty subtries are actually present, Figure 2 (a). Width compression can also be combined with path compression, Figure 2 (b).

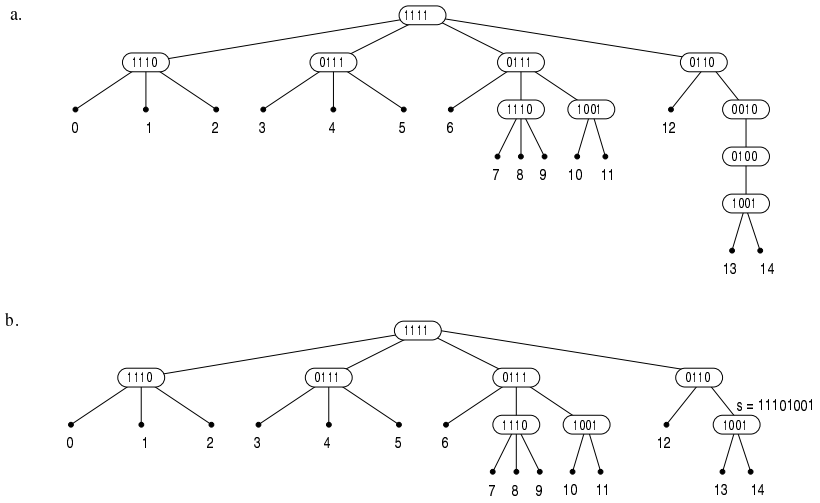


Figure 2: A width-compressed quaternary trie (a), and a path- and width-compressed quaternary trie (b). The data set of Figure 1 is used. Width compression is realized by a bitmap stored in each node. 1 stands for a non-empty child and 0 for an empty child. The ordering of bits indicates the ordering of non-empty children. In (b), variable s gives the longest common prefix string of a path-compressed subtrie. As before, the labels $0 - 14$ only indicate the presence of an element.

The primary goal of level compression is to reduce the depth of the trie. We can also apply level compression to a path- and width-compressed trie as long as we keep the cost of copying reasonable, Figure 3. Note that in a path- and width-compressed trie level compression is applied to sparsely populated nodes. This is quite contrary to the way level compression works in the path- and level-compressed trie where only

complete nodes are level-compressed. In the path-, width-, and level-compressed trie we start with a trie of the largest possible out-degree because width compression removes all empty subtries.

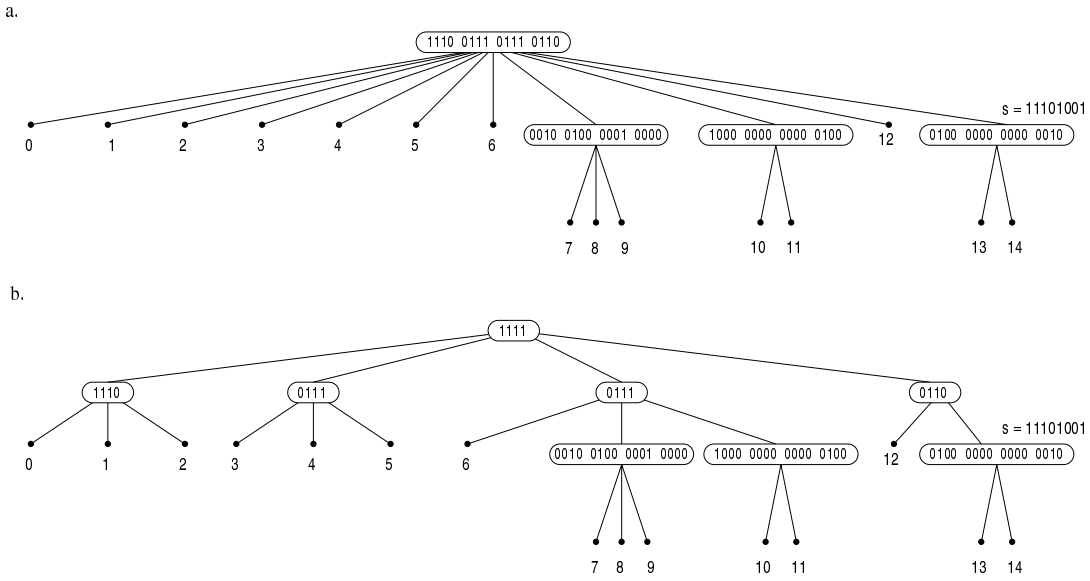


Figure 3: A path- and width-compressed hexadecimal trie (a), and the corresponding path-, width- and level-compressed trie (b). The data set of Figure 1 is used. In trie (b), the root of trie (a) is transformed into a cluster of quaternary nodes to reduce the cost of copying. As before, the labels $0-14$ only indicate the presence of an element.

Compression methods for functional tries are hardly discussed in literature. Goubalt [13] describes a system-level implementation of a functional binary trie and uses the trie for introducing sets and maps into a functional set-oriented programming language. He does not apply any compression methods to the trie, however. Okasaki [22] shows how maps are used in a functional programming language to implement tries. Even though maps can be viewed as a means to implement width compression, they do not bring about the savings in memory utilization we are striving for. To our knowledge, this is the first experimental study comparing efficient system-level compression methods for functional tries.

2 Compression methods for functional tries

In this section we give a brief overview of the compression techniques we use for functional tries. We start with the definition of a binary trie depicted in Figure 1 (a). It contains strings from the alphabet $\sum = \{0, 1\}$. We say that a string w is the i-*suffix* of the string u, if there is a string v of length i such that $u = vw$.

Definition 1 *A* binary trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is an empty tree.*
- *If $n = 1$, the trie consists of one node representing the element.*
- *If $n > 1$, the trie consists of a node of two children. For each $c \in \{0, 1\}$ there is a child, which is a binary trie and contains the 1-suffixes of all elements starting with c.*

We assume that all strings in a trie are prefix-free: no string can be a prefix of another. In particular, this implies that duplicate strings are not allowed. If all strings stored in the trie are unique, it is easy to ensure that the strings are prefix-free by appending a special marker at the end of each string. For example, we can append the string 1000... to the end of each string. A finite string that has been extended in this way is often referred to as a semi-infinite string or sistring.

A path-compressed binary trie, Figure 1 (b), is a trie where all subtries with a single non-empty child have been removed. The path info corresponding to the removed single-child subtries is represented as a digit string.

Definition 2 *A* path-compressed binary trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is an empty tree.*
- *If $n = 1$, the trie consists of one node representing the element.*
- *If $n > 1$, the trie consists of a node of two non-empty children and a binary string s of length $|s|$. This string is the longest prefix common to all elements stored in the trie. For each $c \in \{0, 1\}$ there is a child, which is a path-compressed binary trie and contains the $|s| + 1$-suffixes of all elements starting with sc.*

The next step is to apply level compression to a path-compressed binary trie. The resulting trie is a multi-digit trie called the path- and level-compressed trie (see Figure 1 (b)), and it is defined as follows.

Definition 3 *A* path- and level-compressed trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is an empty tree.*
- *If $n = 1$, the trie consists of one node representing the element.*
- *If $n > 1$, the trie consists of a node of 2^i children where $i \geq 1$ is maximal such that all children are non-empty and a binary string s of length $|s|$. The string is the longest prefix common to all elements stored in the trie. For each binary string x of length i there is a child, which is a path- and level-compressed trie and contains the $(|s| + |x|)$-suffixes of all elements starting with sx.*

A indicated by the definition above, there are no empty children in a path- and level-compressed trie storing more than one element. If empty children are allowed in the data structure it is called the *relaxed* path- and level-compressed trie.

Width compression is yet another method for removing empty children from a trie. In a width-compressed trie depicted in Figure 2 (a) only non-empty children are present.

Definition 4 *A* width-compressed trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is an empty tree.*
- *If $n = 1$, the trie consists of one node representing the element.*
- *If $n > 1$, the trie consists of a node of 2^k children for some fixed $k \geq 1$. For each binary string x of length k there is a child that is a width-compressed trie, but only non-empty children are represented in the node. Let $d[0], \ldots, d[2^k - 1]$ be the sequence of all children and $e[0], \ldots, e[m], m \in [0, 2^k - 1]$ the sequence of non-empty children. Each non-empty $d[i]$ maps injectively and order-preservingly to some $e[j]$.*

Path compression can be combined with width compression, see Figure 2 (b). The path- and width-compressed trie is defined as follows.

Definition 5 *A* path- and width-compressed trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is an empty tree.*
- *If $n = 1$, the trie consists of one node representing the element.*
- *If $n > 1$, the trie consists of a node of 2^k children for some fixed $k \geq 1$ and a binary string s of length $|s|$. The string is the longest prefix common to all elements stored in the trie. For each binary string x of length k there is a child, which is a path- and width-compressed trie and contains the $(|s| + |x|)$-suffixes of all elements starting with sx. Only non-empty children are represented in the node. Let $d[0], \ldots, d[2^k - 1]$ be the sequence of all children and $e[0], \ldots, e[m], m \in [1, 2^k - 1]$ the sequence of non-empty children. Each non-empty $d[i]$ maps injectively and order-preservingly to some $e[j]$.*

If we choose $k = 1$ in the previous definition we get the path-compressed binary trie.

The path-, width-, and level-compressed trie generalizes the definition of the path- and width-compressed trie by allowing trie nodes or varying out-degree, see Figure 3.

Definition 6 *A* path-, width-, and level-compressed trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is an empty tree.*
- *If $n = 1$, the trie consists of a node representing the element.*
- *If $n > 1$, the trie consists of a node of 2^i children for some $i \geq 1$ and a binary string s of length $|s|$. The string is the longest prefix common to all elements stored in the trie. For each binary string x of length i there is a child, which is a path-, width-, and level-compressed trie and contains the $(|s| + |x|)$-suffixes of all elements starting with sx. Only non-empty children are represented in the node. Let $d[0], \ldots, d[2^i - 1]$ be the sequence of all children and $e[0], \ldots, e[m], m \in [1, 2^k - 1]$ the sequence of non-empty children. Each non-empty $d[j]$ maps injectively and order-preservingly to some $e[k]$.*

As indicated by our simple analysis in Section 1, a quaternary trie minimizes the cost of copying. To avoid binary nodes, we have to define $i = 2j, j \geq 1$. Width compression can be applied to tries of arbitrary degrees. The choice of out-degrees dependents on the method that is used for implementing the set of labels for non-empty children and the injective and order-preserving mapping from $d[i]$'s to $e[j]$'s. As shown in Figure 3, we use a bitmap for the set of labels of non-empty children. The bitmap is part of one-word administrative data. In our system, eight bits are reserved for low-level type info. As we have only 24 bits at our disposal in a 32-bit machine architecture, the upper limit to the out-degree is 16 – a hexadecimal subtrie.

3 Implementation of compression methods

When implementing a functional path-, width-, and level-compressed trie, we are confronted with the following issues. First we have to specify the out-degrees 4^i we are able to support. The mere out-degree is not enough, however, because we want to keep the copy cost reasonable. Thus, an additional parameter – the upper bound for non-empty children – must be specified for each out-degree. Then we have to find an efficient method for the manipulation of bitmaps in width-compressed nodes. When using a bitmap representation, it is simple to specify if a child is empty or non-empty. It is less obvious how to efficiently compute the location of a non-empty child in an array storing only non-empty children. Finally, we have to find the most suitable representations for variable-size trie nodes and longest common prefix strings.

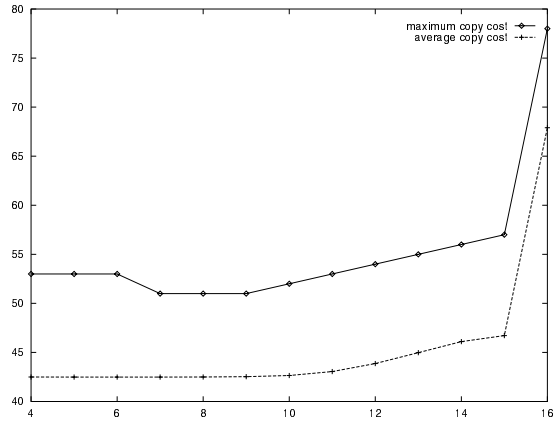


Figure 4: The average and maximum costs of copying as a function of the upper bound for the number of children.

We have implemented our data structure in C on top of Shades. Shades organizes data in cells, each having a one word header used for administrative data. In the header eight bits are reserved for low-level type info devoted to the internal use of the memory manager. We have restricted the out-degrees of our trie to 4 and 16 and set the upper bounds for non-empty children to 4 and 13, respectively. The number of children specify the low-level type of a node. For example, there are three low-level types for a quaternary node, one having 2, one 3, and one 4 children.

We have specified the upper bounds for non-empty children experimentally. To find the average and maximum costs of copying, we ran update tests for a hexadecimal trie with different upper bounds for non-empty children using 100,000 unique uniform random keys, LCG(2^{32}, 2147001325, 715136305, 0), see Figure 4. Note the abrupt increase in the average and the maximum costs of copying when full hexadecimal nodes are allowed. This increase is distribution dependent and caused by full nodes at the upper levels of the trie when storing random keys.

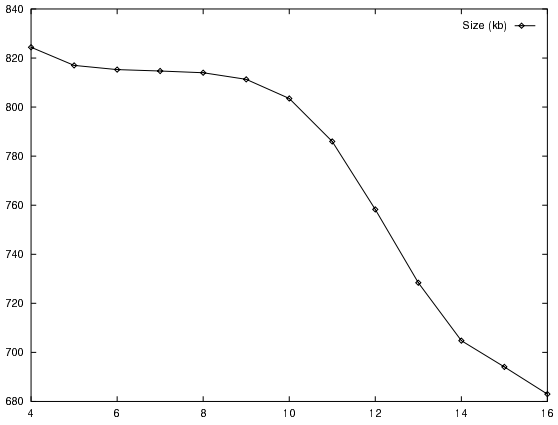


Figure 5: The size of the trie as a function of the upper bound for the number of children.

The same data set was used to measure the overall memory consumption for each upper bound, see Figure 5. We have chosen 13 as the upper bound for a hexadecimal node, because it provides a good trade-off between the cost of copying and the overall memory consumption. Another reason for our choice was that a hexadecimal node of 14 children gives rise to a cluster of quaternary nodes where the father is full and each of its children has at least two grandchildren. Thus, none of the children in the cluster is path-compressed. This speeds up searches within the cluster of quaternary nodes because the path compression checks and the node type checks can be eliminated altogether, see the pseudo-code for the function PWL-LOOKUP below.

```
PWL-LOOKUP(p : Trie, key : key to be searched)
 1  if p is a hexadecimal node then
 2     n ← The number of bits in the longest common prefix of p.
 3     i ← The Nat value of four bits succeeding the most significant n bits in key.
 4     q ← p.children(MAP(p.bitmap, i))
 5     PWL-LOOKUP(q, key)
 6  else
 7     if p is a quaternary node then
 8        n ← The number of bits in the longest common prefix of p.
 9        i ← The Nat value of two bits succeeding the most significant n bits in key.
10        q ← p.children(MAP(p.bitmap, i))
11        j ← The Nat value of two bits succeeding the most significant n + 2 bits in key.
12        r ← q.children(MAP(q.bitmap, j))
13        PWL-LOOKUP(r, key)
14     else
15        if p is a node representing the element and p is equal to key  then
16           p
17        else
18           NIL
```

Yet another reason for restricting the nominal out-degree to 16 is efficient implementation of the set of labels for non-empty children and the mapping from all children to non-empty children. Recall that eight bits of the one word cell header are used for low-level typing. In the 32-bit architecture, the remaining 24 bits of the header are at our disposal for other purposes. We use 16 bits for a bitmap where a set bit indicates the existence of a non-empty child with a digit equal to the bit position. Now, only non-null pointers need to be stored. The order of the pointers is given by order of bits in the bitmap.

The bitmap is used for computing the relative position of a pointer within the child pointer array. An array of 2^{16} elements would be required to represent all the positions of the possible child pointer combinations in a hexadecimal node. However, we can do the computation piecewise – eight bits at a time – by using a smaller array of 2^8 elements. One array lookup is needed to compute the relative position of a pointer in a width-compressed quaternary node. One lookup or two lookups and an addition are required for a width-compressed hexadecimal node, see the pseudo-code for the function MAP below.

```
MAP(bitmap : 0 ... 2^16 − 1, ind : 0 ... 15)
1  mapArray : A 256-element array constant mapping logical array indexes to physical ones.
2  if ind < 8 then
3     mapArray(bitmap MOD 2^ind)
4  else
5     mapArray(bitmap MOD 2^8 + (bitmap DIV 2^8) MOD 2^(ind−8))
```

We are still left with one implementation issue: how to represent the longest common prefix strings. We have solved this by merely storing the length of the longest common prefix string. Note that we need the value of the longest common prefix string in updates only. In searches, it is cheaper to search further and perform a single key comparison in the leaf. Thus, there is always one key comparison per search. This strategy manifest itself in the MAP function, too, as no checks are needed for empty children: the labels of empty children are mapped to the closest non-empty children. We have actually chosen a similar strategy for updates, too. When the prefix string is needed we locate one of the leaves of the current node by searching. Of course each node could store a direct pointer to a leaf, but the moderate computational overhead due to leaf searching saves us one word per node.

4 Experimental results

We have compared different compression strategies for functional binary, quaternary, hexadecimal, and multi-digit tries: mere path compression, a combination of path and level compression, and a combination of path, width and level compression. In the functional multi-digit trie, we have restricted the nominal out-degrees to four and sixteen due to the reasons explained in the previous section. The multi-digit trie is the best choice for a functional trie, as it provides a good compromise between the memory requirements, the cost of copying in updates, and the average and maximum path lengths.

4.1 Method

The data structures have been implemented on top of the persistent memory manager Shades. Shades supports disk-backed automatic memory management in a soft real-time environment. Recovery is based on shadowing combined with real-time generational stop-and-copy garbage collection. As shadowing relies upon path copying, recovery does not impose any additional overhead to the implementation of functional data structures when disk writes are eliminated.

It's always problematic to conduct performance measurements in an environment supporting automatic memory management. In updates, there is no exact way of specifying the overhead of automatic memory management. We have used relatively large memory sizes to minimize the overhead of the mature generation garbage collection. We have also directed the output to `/dev/null` to avoid the overhead of disk writes.

We have implemented six trie structures: the path-compressed binary trie (P-2), the path-compressed quaternary trie (P-4), the path- and width-compressed quaternary trie (PW-4), the path-compressed hexadecimal trie (P-16), the path- and width-compressed hexadecimal trie (PW-16), and the path-, width-, and level-compressed trie (PWL). In the tables below, the tries are identified by the shorter names given in parentheses. For each trie we give its size, its average and maximum path lengths, its average and maximum copy costs per update, and the number of searches and updates per second. The size figures do not include the memory consumption of leaves, because it is equal in all tries.

We have chosen the following test data sets for our experimental study. The first data set is synthetic and consists of 100,000 unique keys generated using a linear congruence random number generator $LCG(2^{32}, 2147001325, 715136305, 0)$ (Table 1). The other data sets are real data: binary strings from Internet routing tables (Table 2), two-dimensional geometric point data (Table 3) visualized in Figure 6 from [27], and ASCII strings from the Calgary Text Compression Corpus, which is a standardized text corpus used frequently in data compression research (Table 4).

The performance tests were measured in the Linux 2.0 operating system environment running on a Pentium Pro-S 200 MHz server with a 256 kB internal cache. Before doing the actual measurements we first populated the data structures and created a table of 50,000 elements that were randomly chosen from the data sets in question. Then each test was run for two minutes, and the table of random elements was repeatedly

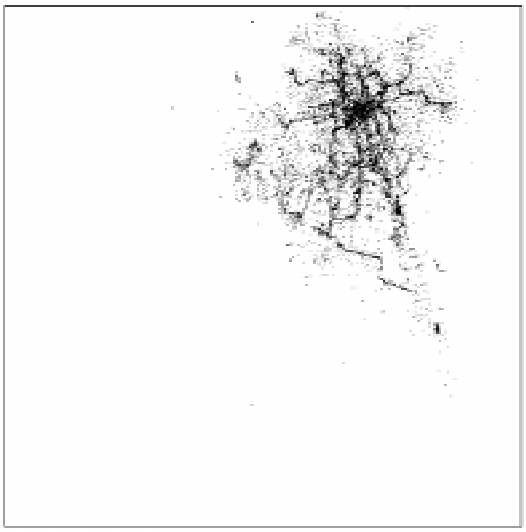


Figure 6: Drill holes in Munich.

used as input. Note that the update figures do not include inserts and deletes. The cost of inserts and deletes is very close to the cost of updates because inserts and deletes do not induce any major reorganizations in functional trie structures.

Trie	*Size*	*Depth*		*Copy Cost*		*Searches*	*Updates*
	(kB)	*Aver*	*Max*	*Aver*	*Max*	*per sec*	*per sec*
P-2	1,171.9	16.94	21	50.83	63	327,118	122,926
P-4	1,214.4	9.01	12	45.07	60	441,306	146,413
PW-4	876.4	9.01	12	**42.69**	**53**	421,230	147,275
P-16	2,485.3	**4.85**	**7**	82.51	119	521,648	78,833
PW-16	**683.0**	**4.85**	**7**	67.90	78	**528,541**	118,092
PWL	728.4	8.19	11	44.98	55	436,681	**147,907**

Table 1. Uniform random (100,000 unique entries).

Trie	*Size*	*Depth*		*Copy Cost*		*Searches*	*Updates*
	(kB)	*Aver*	*Max*	*Aver*	*Max*	*per sec*	*per sec*
P-2	449.6	18.56	24	55.68	72	376,223	112,322
P-4	438.4	9.91	13	49.57	65	524,384	129,299
PW-4	325.2	9.91	13	**46.62**	**61**	476,872	129,702
P-16	730.8	**5.23**	**7**	88.92	119	**660,066**	82,535
PW-16	**235.8**	**5.23**	**7**	71.75	101	625,782	118,301
PWL	247.7	8.51	12	49.26	70	508,647	**141,965**

Table 2. Mae-east routing table (38,470 entries, 38,367 unique entries).

Trie	*Size*	*Depth*		*Copy Cost*		*Searches*	*Updates*
	(kB)	*Aver*	*Max*	*Aver*	*Max*	*per sec*	*per sec*
P-2	228.0	17.08	23	51.23	69	355,366	126,119
P-4	236.2	9.58	13	47.92	65	497,265	146,349
PW-4	170.5	9.58	13	43.51	**58**	451,060	148,192
P-16	443.9	**5.12**	**7**	87.11	119	**618,429**	94,366
PW-16	**128.2**	**5.12**	**7**	62.15	81	550,964	140,706
PWL	136.5	7.89	11	**42.99**	60	475,737	**165,289**

Table 3. Drill holes (19,461 unique entries).

Trie	*Size*	*Depth*		*Copy Cost*		*Searches*	*Updates*
	(kB)	*Aver*	*Max*	*Aver*	*Max*	*per sec*	*per sec*
P-2	193.8	20.36	41	61.07	123	253,100	89,421
P-4	272.1	14.49	30	72.46	150	282,247	83,181
PW-4	173.5	14.49	30	54.00	109	261,917	91,475
P-16	759.2	**9.92**	**19**	168.64	323	278,396	40,496
PW-16	**153.9**	**9.92**	**19**	57.38	115	**300,030**	99,552
PWL	154.3	10.53	20	**53.56**	**108**	274,650	**104,254**

Table 4. English text (16,541 unique entries).

4.2 Discussion

The test results demonstrate the benefits of applying all of the compression methods together. Path compression and width compression produce the most significant savings in memory utilization and cost of copying. Level compression further reduces the size of the data structure and its average depth without a substantial increase in the copy cost. The way level compresssion and width compression interact is best demonstrated by the "Drill holes" and "English text" test cases. Even though the number of drill holes is greater than the number of text items, width compression and level compression pack the highly clustered drill holes so efficiently that the size of the "Drill holes" trie is smaller than the size of the "English text" trie.

As stated in the previous section we used large memory sizes to eliminate the overhead of the mature generation garbage collection. In real applications, however, free memory may become exhausted. When mature generations are also collected, the size of a data structure should have a direct impact on the overall performance. Because our measurements did not take this cumulative cost of copying into account, we ran a main-memory variant of the TPC-B benchmark [14] to investigate the effect of the collection of mature generations using Patricia trees, path- and width-compressed quaternary tries, and path-, width- and level-compressed tries. In the tests we did not find any significant deviations from our earlier results.

We have not included balanced trees in this experimental study. Our experiences of running the AVL-tree variant of the main-memory TPC-B benchmark indicate that tries outperform AVL-trees in terms of memory utilization, copy cost and update and search performance. The AVL-tree is not an optimal choice for a functional balanced tree, however. We find the degree k minimizing the approximate copy cost of a balanced tree by repeating the simple analysis we did in the first chapter for tries. As previously, we assume that all leaves are at the same level in the tree. Thus, our tree is an external balanced tree. The tree consists of internal nodes holding $2k-1$ pointers: k pointers to subtrees and $k-1$ key pointers servings as routers, and N leaves: the stored keys. The approximate copy cost is given by $f(k) = (2k-1+a)\log_k N$ where a is the size of administrative data in a tree node. With $a = 1$ the minimum of $f(k)$ is given by $f'(k) = 0$, i.e. $\ln k = 1$ or simply $k = e$. The discrete values closest to k are 2 and 3, and $f(2) > f(3)$ for all $N > 1$. Thus, the external 2-3 tree minimizes the copy cost. Interestingly, the analysis above applies even to internal balanced trees assuming a large N.

External 2-3 tree, bucket capacity 5	*Appr Aver Size (kB)*	*Appr Aver Depth*	*Appr Aver Copy Cost*
Uniform random	755.0	9.35	41.75
Mae-east routing table	289.7	8.48	37.85
Drill holes	146.9	7.86	35.09
English text	124.9	7.71	34.43

Table 5. The approximate size, the approximate average depth, and the approximate average copy cost for external 2-3 trees with bucket capacity 5.

Internal 2-3 tree	*Appr Aver Size (kB)*	*Appr Aver Depth*	*Appr Aver Copy Cost*
Uniform random	1,258.3	9.81	41.59
Mae-east routing table	482.8	8.94	37.70
Drill holes	244.9	8.32	34.94
English text	208.1	8.18	34.28

Table 6. The approximate size, the approximate average depth, and the approximate average copy cost for internal 2-3 trees.

In Tables 5 and 6 we give approximations for the average depth, the average copy cost and the overall storage utilization of the external 2-3 tree with bucket capacity 5 and the internal 2-3 tree, respectively, when they are used for storing our test data. The bucket capacity 5 makes the size of the bucket node equal to the size of the internal node. We also take into account the effect of variable-size nodes by using the scale factor $\ln 2$ in computing the copy cost and storage requirements. The scale factor $\ln 2$ gives the approximate average storage utilization of hierarchical access methods based on the binary splitting policy [3]. All the formulas used for computing the values are given in the appendix.

It is clear that approximations cannot be taken for measured results. For balanced trees, however, approximations give a fairly reliable estimate of the actual results. In all test cases, the measured sizes of the PWL-trie are closer to the approximated sizes of the external 2-3 tree than to those of the internal 2-3 tree. Excluding the "English text" test case, the measured average depths are closer to the approximated average depths of the external 2-3 tree than to those of the internal 2-3 tree. Excluding uniform random data, the measured copy costs of the PWL trie are somewhat greater than the approximated copy costs of both the internal and the external 2-3 tree. This is probably due to better balance in the balanced trees. The differences are not significant, however.

Is is difficult to draw any reliable conclusions about the performance of the PWL-trie compared to 2-3 trees. Some observations affecting the performance can be made, however. PWL-trie lookups are extremely efficient, because only a single key comparison is needed per lookup. In inserts and deletes, all modifications of the PWL-trie are local within the scope of a hexadecimal node. No global operations such as balancing are needed.

5 Conclusions

In our experimental study we developed and compared different compression methods for functional tries. All of the compression methods have been previously used for imperative tries, but their application to functional tries is hardly discussed in literature. In addition to applying compression methods to functional tries, we also developed a new method for efficient implementation of width compression that can even be used for imperative tries.

Six different functional trie structures were designed, implemented, and tested using both synthetic and real data: the path-compressed binary trie, the path-compressed quaternary trie, the path- and width-

compressed quaternary trie, the path-compressed hexadecimal trie, the path- and width-compressed hexadecimal trie, and the path-, width- and level-compressed trie. When accounting for the trie size, the average path length, the copy cost, and the search and update performance, the path-, width- and level-compressed trie, or the PWL-trie for short, was found the best. To our knowledge, the PWL-trie is the first functional trie structure where path compression, width compression and level compression have been efficiently combined.

We also compared some of the results of the PWL-trie to approximations calculated for the external 2-3 tree with bucket capacity 5 and the internal 2-3 tree. We chose 2-3 trees because – as we showed – they minimize the copy cost in balanced trees. In all test cases the size and the average depth of the PWL-trie were close to the approximated average size and approximated average depth of the external 2-3 tree. The approximated copy cost in 2-3 trees was smaller than the measured copy cost for the PWL-trie. The difference was not significant, however, and it is probably due to better balance in 2-3 trees. Further experimental research is still needed, however, to verify the results of our approximate analysis.

PWL-trie lookups are extremely efficient, because only a single key comparison is needed per lookup. In inserts and deletes all modifications of the PWL-trie are local with respect to the current node. No global operations such as balancing are needed. Moreover, most operations – especially group operations – are easier to implement in tries than in balanced trees and they generalize to multi-dimensional data. Taking additionally into account the moderate size and copy cost of the PWL-trie, we consider it as an ideal choice for a general-purpose functional main-memory index structure.

Acknowledgements

We are greatly indebted to Kenneth Oksanen for his pioneering work on real-time garbage collection methods for persistent functional heaps. We thank Vera Izrailit, Petri Mäenpää, Eljas Soisalon-Soininen, and the anonymous referees for comments that helped us to improve this paper.

This work has been carried out in the HiBase and HiPro projects, joint research projects of Nokia Telecommunications and Helsinki University of Technology. For more details, please, consult our home page at `http://hibase.cs.hut.fi/`. The projects have been financially supported by the Technology Development Centre of Finland (Tekes).

Appendix: Formulas

The external 2-3 tree with bucket capacity 5

1. The approximate average height of the tree: $h = \log_3 \frac{N}{5 \ln 2}$ where N is the number of elements, 5 the bucket capacity, and $\ln 2$ the approximate average storage utilization of a hierarchical access method based on binary splitting.

2. The approximate average copy cost: $h \cdot (5 \ln 2 + 1)$ where h is given above and $5 \ln 2 + 1$ is the average size of the tree node including administrative data.

3. The approximate number of tree nodes: $n = \frac{3^{h+1}-1}{2}$ where h is given above.

The internal 2-3 tree

1. The approximate number of tree nodes: $n = \frac{N}{2 \ln 2}$ where N is the number of elements, $\ln 2$ the approximate average storage utilization of a hierarchical access method based on binary splitting, and $2 \ln 2$ the approximate average number of keys in a tree node.

2. The approximate average height of the tree: Solution of the equation $\frac{N}{2 \ln 2} = \frac{3^{h+1}-1}{2}$

3. The approximate average path length of the tree: $l = \frac{\sum_{i=0}^{h} i3^i}{\sum_{i=0}^{h} 3^i}$

4. The approximate average copy cost: $l \cdot (5 \ln 2 + 1)$ where l is given above and $5 \ln 2 + 1$ is the average size of the tree node including administrative data.

References

[1] A. Andersson, S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.

[2] A. Andersson, S. Nilsson. Faster searching in tries and quadtrees – an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 82–93, 1994. LNCS 855.

[3] C-H. Ang, H. Samet. Approximate average storage utilization of bucket methods with arbitrary fanout. *Nordic Journal of Computing*, 3(1996):280-291, 1996

[4] J.-I. Aoe, K. Morimoto. An efficient implementation of trie structures. *Software – Practice and Experience*, 22(9):695–721, 1992.

[5] J.J. Darragh, J.G. Cleary, I.H. Witten. Bonsai: A compact representation of trees. *Software – Practice and Experience*, 23(3):277–291, 1993.

[6] L. Devroye. A note on the average depth of tries. *Computing*, 28(4):367–371, 1982.

[7] J.A. Dundas III. Implementing dynamic minimal-prefix tries. *Software – Practice and Experience*, 21(10):1027–1040, 1991.

[8] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20:345–369, 1983.

[9] P. Flajolet, M. Régnier, D. Sotteau. Algebraic methods for trie statistics. *Ann. Discrete Math.*, 25:145–188, 1985.

[10] P. Flajolet. Digital search trees revisited. *SIAM Journal on Computing*, 15(3):748–767, 1986.

[11] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[12] G.H. Gonnet, R.A. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, second edition, 1991.

[13] J. Goubault. HimML: standard ML with fast sets and maps. *Proc. of the 5th ACM SIGPLAN Workshop on Standard ML and its Applications (ML '94), Orlando, Florida, USA*, 62–69, 1994.

[14] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, San Matea (CA), USA, 1993.

[15] P. Kirschenhofer, H. Prodinger. Some further results on digital search trees. *Proc. 13th ICALP*, pages 177–185. Springer-Verlag, 1986. Lecture Notes in Computer Science vol. 26.

[16] D.E. Knuth. *TEX: The Program*. Addison-Wesley, 1986.

[17] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[18] J. van Leeuwen (ed.). *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[19] K. Morimoto, H. Iriguchi, J.-I. Aoe. A method of compressing trie structures. *Software – Practice and Experience*, 24(3):265–288, 1994.

[20] S. Nilsson, G. Karlsson. Fast address lookup for internet routers. In Paul K editor, *Proceedings of the 4th IFIP International Conference on Broadband Communications (BC'98)*. Chapman & Hall, 1998.

[21] S. Nilsson, M. Tikkanen. Implementing a dynamic compressed trie. In Mehlhorn K editor, *Proceedings of the 2nd Workshop on Algorithm Engineering (WAE'98)*, 25–36, Aug 20-22, 1998.

[22] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[23] K. Oksanen. *Real-time Garbage Collection of a Functional Persistent Heap*. Licentiate's Thesis, Helsinki University of Technology, Faculty of Information Technology, Department of Computer Science, 1999. Also available at `http://hibase.cs.hut.fi`.

[24] B. Pittel. Asymptotical growth of a class of random trees. *The Annals of Probability*, 13(2):414–427, 1985.

[25] B. Pittel. Paths in a random digital tree: Limiting distributions. *Advances in Applied Probability*, 18:139–155, 1986.

[26] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, Jul 1986.

[27] F. Wagner and A. Wolff. A combinatorial framework for map labeling. In Whitesides Sue H. (ed.), *Proceedings of the Symposium on Graph Drawing '98*, Lecture Notes in Computer Science, 1547:616–331, Springer-Verlag, 1998. Also available from `http://www.inf.fu-berlin.de/map-labeling/papers/ww-cfml-98.ps.gz`.